Spring+Spring MVC+MyBatis从零开始学

掌握SSM框架技术，提升SSM整合开发应用系统的能力

本书系统讲解SSM框架的基础知识和整合技术，并通过代码和案例将理论知识升华，使读者学习完本书以后就能比较全面地掌握SSM框架，并具备使用SSM框架开发应用系统的能力。

吴为胜 杨章伟 著



清华大学出版社



Spring+Spring MVC +MyBatis从零开始学

吴为胜 杨章伟 著

清華大學出版社
北京

内 容 简 介

本书详细地讲解 Java EE 中 Spring、Spring MVC 和 MyBatis 三大框架（以下简称 SSM）的基础知识和应用。本书由浅入深、循序渐进，在对知识点进行讲解时采用大量案例，可以更好地帮助读者学习和理解 SSM 的核心技术。本书重点章节对应提供了 37 个教学视频。

本书共 16 章，分为 4 部分。第一部分是 Spring 篇，涵盖 Spring 基础、Spring 中的 Bean、Spring AOP、Spring 的数据库开发、Spring 的事务管理。第二部分是 MyBatis 篇，涵盖初识 MyBatis、MyBatis 的核心配置、动态 SQL、MyBatis 的关联映射和 MyBatis 与 Spring 的整合。第三部分是 Spring MVC 篇，涵盖 Spring MVC 入门、Spring MVC 数据绑定、JSON 数据交互和 RESTful 支持、拦截器。第四部分是应用实战篇，包含 SSM 框架整合和完整的 SSM 实战案例。

本书内容精练、重点突出、实例丰富，是 SSM 框架技术学习者的必备参考书，同时非常适合大中专院校师生学习阅读，也可作为高等院校计算机专业的教材使用。

图书在版编目（CIP）数据

Spring+Spring MVC+MyBatis 从零开始学/吴为胜，杨章伟著.—北京：清华大学出版社，2019
ISBN 978-7-302-52871-5

Ⅰ. ①S… Ⅱ. ①吴… ②杨… Ⅲ. ①JAVA 语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2019）第 083028 号

责任编辑： 夏毓彦
封面设计： 王　翔
责任校对： 闫秀华
责任印制：

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：
经　　销： 全国新华书店
开　　本： 190mm×260mm　　**印　　张：** 17　　**字　　数：** 435 千字
版　　次： 2019 年 7 月第 1 版　　**印　　次：** 2019 年 7 月第 1 次印刷
定　　价： 59.80 元

产品编号：081775-01

前　　言

目前 Java EE 应用开发通常会采用框架技术，使得 Java EE 架构具有高度的可维护性和可扩展性，同时极大地提高了项目的开发效率、降低了开发和维护成本。本书主要讲解的 SSM（Spring+Spring MVC+MyBatis）框架技术注重注解开发、ORM 实现灵活、SQL 优化简便，学习容易入门。现实生活中对性能要求较高的项目通常会选用 SSM 框架。

为了方便广大读者学习，作者结合自己多年的项目开发和培训经验写作本书。本书全面地介绍 Spring、Spring MVC、MyBatis 及三者的整合技术，并配以大量实例贯穿讲解。本书最后一章选取新闻发布管理系统作为综合案例讲解，先对系统做整体分析，再通过对开发过程以及知识点的详细讲解，使读者可以真正具备使用 SSM 框架开发实际项目的能力。

为方便初学者阅读，本书特做如下安排：

- 采用从易到难、循序渐进的方式进行讲解。
- 书中案例采用分步骤实现，让开发过程一目了然。
- 知识点匹配大量实例（含源代码）。
- 重点章节对应提供了 37 个教学视频。

本书内容

本书知识点规划如下（从逻辑上划分为 4 部分）：

第一部分　Spring 篇（第 1~5 章）

讲述 Spring 的基本知识和应用，其中包括 Spring 基础、Spring 中的 Bean、Spring AOP、Spring 的数据库开发、Spring 的事务管理。

第二部分　MyBatis 篇（第 6~10 章）

讲述 MyBatis 的相关知识，其中包括初识 MyBatis、MyBatis 的核心配置、动态 SQL、MyBatis 的关联映射和 MyBatis 与 Spring 的整合。

第三部分　Spring MVC 篇（第 11~14 章）

讲述 Spring MVC 的相关知识，其中包括 Spring MVC 入门、Spring MVC 数据绑定、JSON 数据交互和 RESTful 支持、拦截器。

第四部分　应用实战篇（第 15~16 章）

讲述 SSM 框架的整合与综合应用，其中第 15 章讲述 SSM 框架整合，第 16 章讲述 SSM 实战——新闻发布管理系统的分析、设计与实现。

读者对象

本书适合以下读者翻阅和学习：

- 希望掌握 SSM 框架开发的 Java 新手。
- 使用 Java 开发网络应用的人员。
- 自学制作网站的网络爱好者。
- 构建高性能电子商务系统的开发人员。
- 高等院校和培训学校的师生。

代码、课件、教学视频与相关软件包下载

本书配套代码、课件、教学视频与相关软件包请扫描二维码获取。如果下载有问题，或者对本书有建议与疑问，请联系 booksaga@163.com，邮件主题为“SSM 从零开始学”。



本书作者

本书由萍乡学院吴为胜、杨章伟共同创作完成。

关于封面照片

封面照片由蜂鸟网的摄影家 ptwkzj 先生友情提供，在此表示衷心感谢。

著　者

2019 年 5 月

目　录

第 1 章

Spring 基础

Spring 是目前非常流行的 Java Web 开发框架，用于解决企业应用的复杂性问题。对于一个开发企业级应用的程序员来说，掌握 Spring 框架是必备技能之一。本章主要讲解 Spring 的基础知识。

本章主要涉及的知识点如下：

- Spring 概述：了解 Spring 的基础知识。
- IoC/DI：掌握 Spring 的控制反转/依赖注入的概念，并通过示例熟悉如何实现 IoC/DI。

1.1 Spring 概述

本节首先介绍 Sping 是一个什么样的框架；然后介绍 Spring 框架支持包和相关文件的获取方式，并介绍其目录结构。学习这些的目的是为使用 Spring 框架打好基础。

1.1.1 什么是 Spring

Spring 是一个以 IoC（Inversion of Control，控制反转）和 AOP（Aspect Oriented Programming）为内核的框架。IoC 是 Spring 的基础。IoC 实现的是一种控制，简单地说，就是以前调用 new 构造方法来创建对象，现在变成了使用 Spring 来创建对象。DI（Dependency Inject，依赖注入）与 IoC 的含义相同，从两个角度描述同一个概念。简单地说，DI 就是对象的属性，已经被注入好相关值，直接使用即可。

IoC 和 DI 将在本章后面详细介绍，AOP 将在后续章节详细介绍。

注　意
如果读者是第一次学习本框架，务必严格按照教程的指导，先进行模仿操作，直至看到实际效果。成功之后，再进行改动和调整，从而进一步加深理解，直到熟练掌握。

1.1.2　Spring 的下载及目录结构

Spring 经过十多年的发展，版本不断升级。本书中的实例代码基于 Spring 4.3.6 编写。使用 Spring 框架进行开发需要用到 Spring 框架包和第三方依赖包，具体如下：

1. Spring 框架包

本书中的实例代码基于 Spring 4.3.6 编写，建议读者下载该版本（读者也可以根据实际情况下载最新的版本），其框架压缩包名称为 spring-framework-4.3.6.RELEASE-dist.zip，可以通过地址“http://repo.spring.io/simple/libs-release-local/org/springframework/spring/4.3.6.RELEASE/”下载。下载完成后，将压缩包解压，最终目录结构如图 1.1 所示。

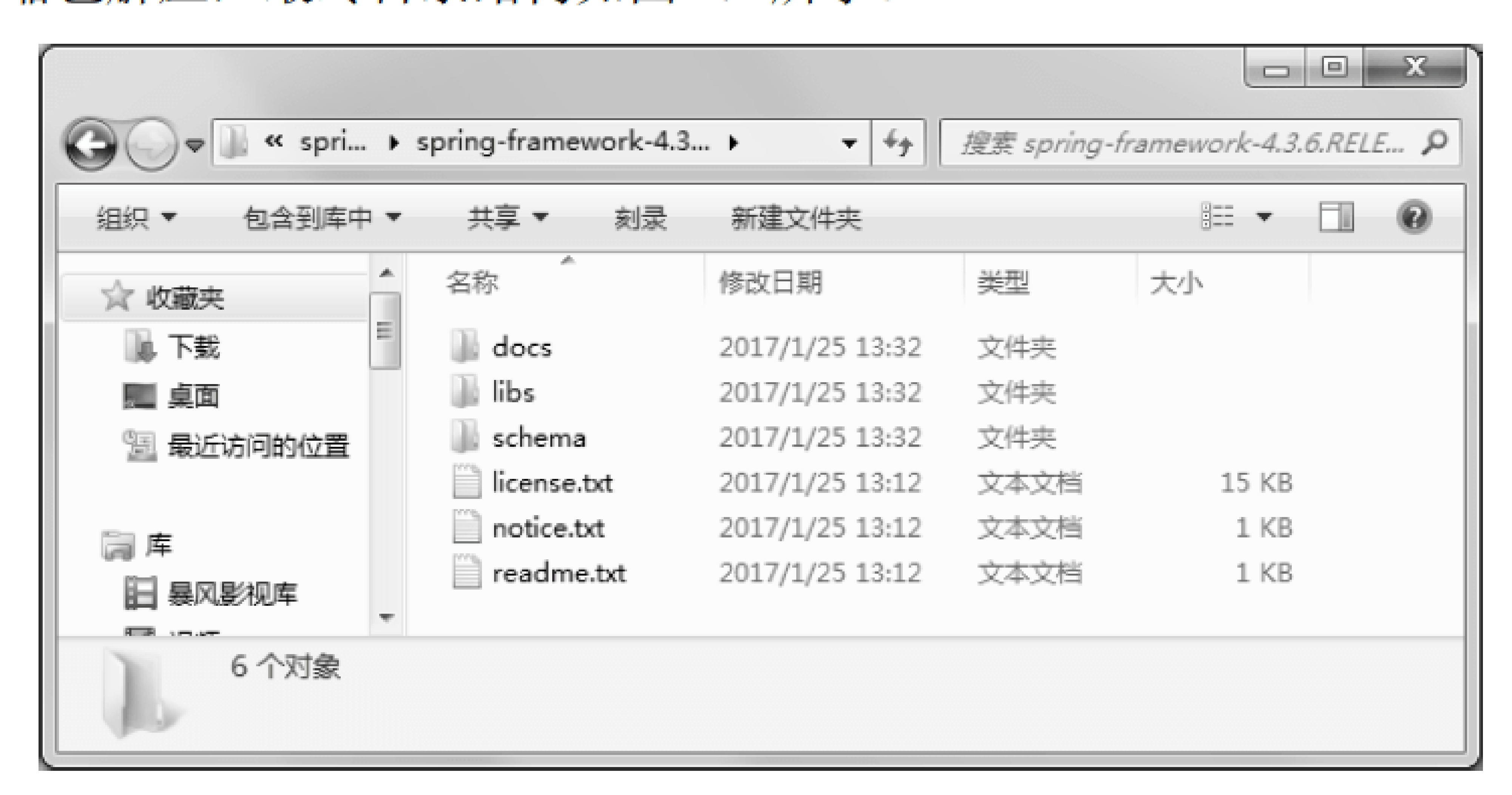


图 1.1　解压后的目录

其中，libs 目录下包含 60 个 JAR 文件，如图 1.2 所示。

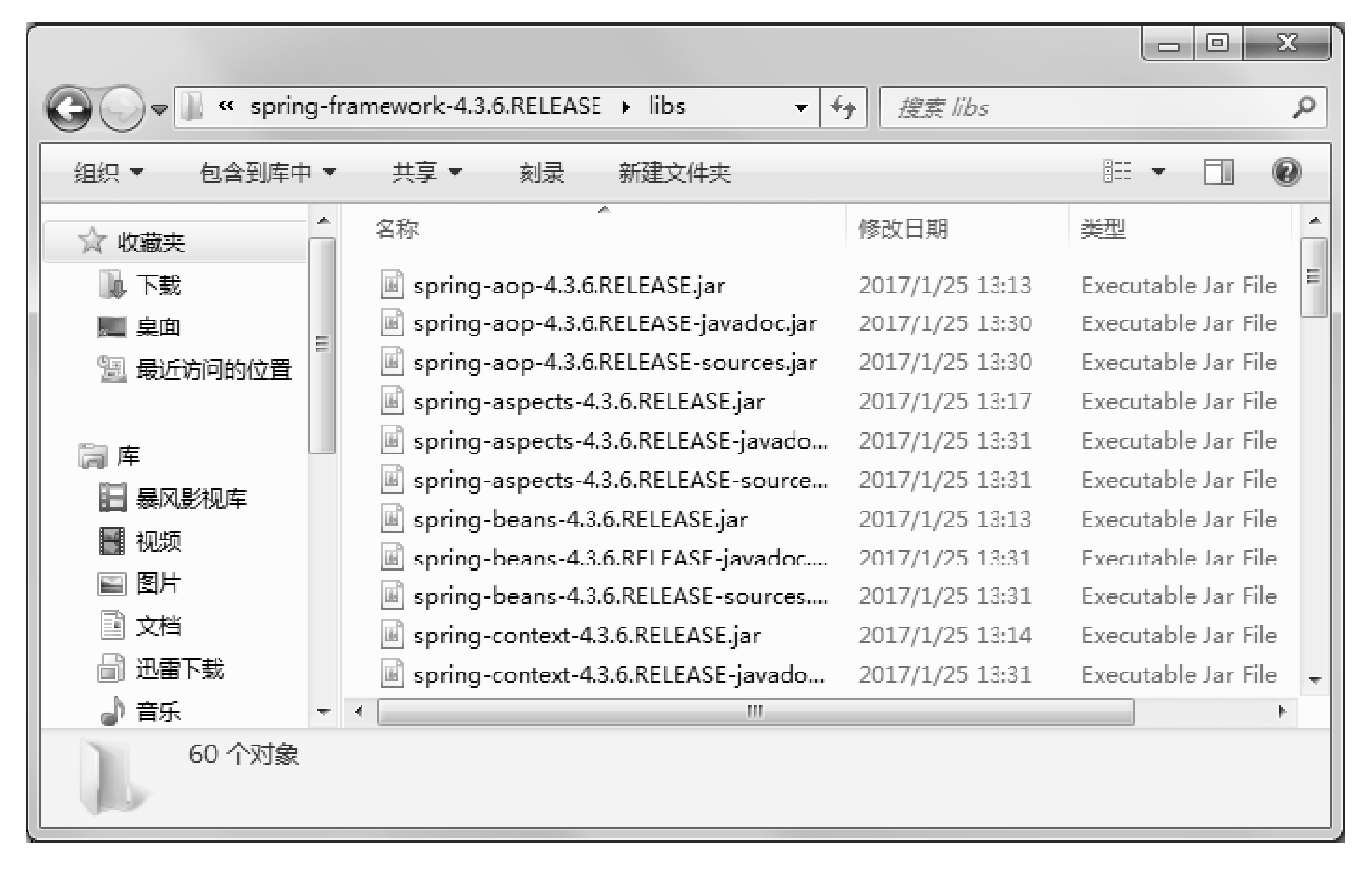


图 1.2　libs 目录

libs 目录中的 JAR 包分为 3 类：

- 以 RELEASE.jar 结尾的是 Spring 框架 class 文件的压缩包。
- 以 RELEASE-javadoc.jar 结尾的是 Spring 框架 API 文档的压缩包。
- 以 RELEASE-sources.jar 结尾的是 Spring 框架源文件的压缩包。

整个 Spring 框架由 20 个模块组成，该目录下 Spring 为每个模块都提供了这 3 类压缩包。

在 libs 目录中，有 4 个 Spring 的基础包，它们分别对应 Spring 核心容器的 4 个模块，具体介绍如表 1.1 所示。

表 1.1　Spring 的基础包说明

包名	说明
spring-core-4.3.6.RELEASE.jar	包含 Spring 框架基本的核心工具类，Spring 其他组件都要用到这个包里的类
spring-beans-4.3.6.RELEASE.jar	所有应用都要用到的 JAR 包，包含访问配置文件、创建和管理 Bean 以及进行 IoC 或者 DI 操作相关的所有类
spring-context-4.3.6.RELEASE.jar	Spring 提供了在基础 IoC 功能上的扩展服务，还提供了许多企业级服务的支持，如任务调度、JNDI 定位、EJB 集成、远程访问、缓存、邮件服务以及各种视图层框架的封装等
spring-expression-4.3.6.RELEASE.jar	定义了 Spring 的表达式语言

2. 第三方依赖包

在使用 Spring 进行开发时，Spring 的核心容器还需要依赖 commons.logging 的 JAR 包。该 JAR 包可以通过网址 http://commons.apache.org/proper/commons-logging/download_logging.cgi 下载。下载后得到一个名为 commons-logging-1.2-bin.zip 的压缩包，将其解压后可以找到 commons-logging-1.2.jar。

> **注　意**
>
> 初学者学习 Spring 框架时，只需将 Spring 的 4 个基础包以及 commons-logging-1.2.jar 包复制到项目的 lib 目录下，并发布到类路径中即可。

1.2　控制反转（IoC）与依赖注入（DI）

IoC 和 DI 是 Spring 的基础，为什么前面提到 IoC 和 DI 描述的是同一概念呢？本节将揭晓答案。

1.2.1　什么是控制反转（IoC）

IoC 是 Inversion of Control 的缩写，译为“控制反转”，还有的译为“控制反向”或者“控制倒置”。

在面向对象传统编程方式中，获取对象的方式通常是用 new 关键字主动创建一个对象。Spring 中的 IoC 方式对象的生命周期由 Spring 框架提供的 IoC 容器来管理，直接从 IoC 容器中获取一个对象，控制权从应用程序交给了 IoC 容器。

IoC 理论上是借助于“第三方”实现具有依赖关系对象之间的解耦，如图 1.3 所示，即把各个对象类封装之后，通过 IoC 容器来关联这些对象类。这样对象与对象之间就通过 IoC 容器进行联系，而对象与对象之间没有什么直接联系。

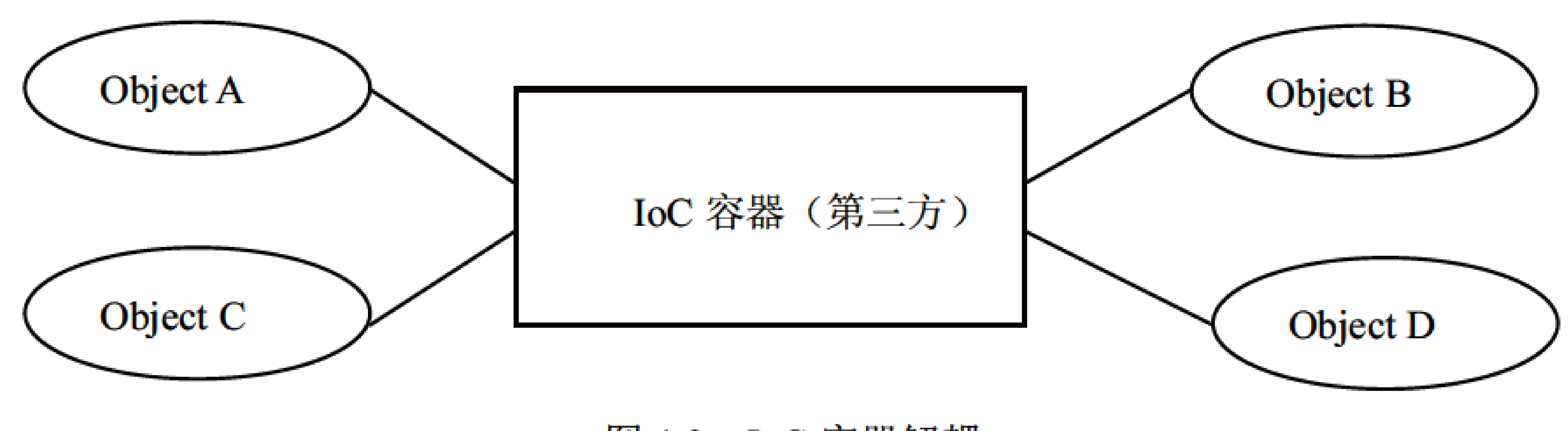


图 1.3 IoC 容器解耦

应用程序在没有引入 IoC 容器之前，对象 A 依赖对象 B，那么 A 对象在实例化或者运行到某一点的时候，自己必须主动创建对象 B 或者使用已经创建好的对象 B，其中无论是创建还是使用已创建的对象 B，控制权都在应用程序自身。如果应用程序引入了 IoC 容器之后，对象 A 和对象 B 之间失去了直接联系，那么当对象 A 实例化和运行时，如果需要对象 B，IoC 容器就会主动创建一个对象 B 注入（依赖注入）到对象 A 所需要的地方。由此，对象 A 获得依赖对象 B 的过程，由主动行为变成被动行为，即把创建对象交给了 IoC 容器处理，控制权颠倒过来了，这就是所谓的控制反转。

1.2.2 什么是依赖注入（DI）

DI 是 Dependency Inject 的缩写，译为“依赖注入”。所谓依赖注入，就是由 IoC 容器在运行期间动态地将某种依赖关系注入对象之中。例如，将对象 B 注入（赋值）给对象 A 的成员变量。

事实上，依赖注入（DI）和控制反转（IoC）是对同一件事情的不同描述，从某个方面讲，就是它们描述的角度不同。依赖注入是从应用程序的角度描述，即应用程序依赖容器创建并注入它所需要的外部资源；而控制反转是从容器的角度描述，即容器控制应用程序，由容器反向地向应用程序注入应用程序所需要的外部资源。这里所说的外部资源可以是外部实例对象，也可以是外部文件对象等。

使用 IoC/DI 给软件开发带来了多方面的益处。

（1）可维护性比较好，便于单元测试、调试程序和诊断故障。代码中的每一个 Class 都可以单独测试，彼此之间互不影响，只要保证自身的功能无误即可，这就是组件之间低耦合或者无耦合带来的好处。

（2）每个开发团队的成员都只需要关注自己要实现的业务逻辑，完全不用关心其他人的工作进展，因为你的任务跟别人没有任何关系，你的任务可以单独测试，不用依赖于别人的组件，再也不会扯不清责任了。所以，在一个大中型项目中，团队成员分工明确、责任明晰，很容易将一个大的任务划分为细小的任务，开发效率和产品质量必将得到大幅度的提高。

（3）可复用性好，我们可以把具有普遍性的常用组件独立出来，反复应用到项目中的其他部分，或者是其他项目，当然这也是面向对象的基本特征。显然，IoC 更好地贯彻了这个原则，提高了模块的可复用性。符合接口标准的实现都可以插接到支持此标准的模块中。

（4）生成对象的方式转为外置方式，就是把对象生成放在配置文件中进行定义。这样，当我们更换一个实现子类将会变得很简单，只要修改配置文件就可以了，完全具有热插拔的特性。

1.2.3 IoC/DI 的实现

Spring 框架的主要功能是通过其核心容器来实现的。Spring 框架提供的两种核心容器分别是 BeanFactory 和 ApplicationContext。IoC/DI 通常有 setter（设置）注入和构造方法注入两种实现方式。

注　意

如前所述，依赖注入（DI）和控制反转（IoC）是对同一件事情的不同描述。所以这里讲的 IoC/DI 实现方式其实就是 DI 实现方式。

1. Spring 核心容器

Spring 框架的两个最基本和最重要的包是 org.springframework.beans.factory（该包中的主要接口是 BeanFactory）和 org.springframework.context（该包中的主要接口是 ApplicationFactory）。

Spring IoC 框架的主要组件有 Beans、配置文件 applicationContext.xml、BeanFactory 接口及其相关类、ApplicationContext 接口及其相关类。

（1）Beans 是指项目中提供业务功能的 Bean，即容器要管理的 Bean。Beans 就是一个常见的 JavaBean、Java 类。

（2）在 Spring 中对 Bean 的管理是在配置文件中进行的。在 Spring 容器内编辑配置文件管理 Bean 又称为 Bean 的装配，实际上装配就是告诉容器需要哪些 Bean，以及容器是如何使用 IoC 将它们配合起来的。Bean 的配置文件是一个 XML 文件，可以命名为 applicationContext.xml 或其他，一般习惯使用 applicationContext.xml。

配置文件包含 Bean 的 id、类、属性及其值，包含一个<beans>元素和数个<bean>子元素。Spring IoC 框架可根据 Bean 的 id 从 Bean 配置文件中取得该 Bean 的类，并生成该类的一个实例对象，继而从配置文件中获得该对象的属性和值。常见 applicationContext.xml 配置文件格式如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
              <!-- 将指定类配置给 Spring，让 Spring 创建其对象的实例  -->
      <bean id= "chinese" class="com.ssm.Chinese">
 <!--
<property>元素用来指定需要容器注入的属性；name 指定属性值为 language；ref 指定需要向
language 属性注入的 id，即注入的对象"英语"，该对象由 English 类生成。
-->
```

```
                    <property name="language" ref="英语"></property>
            </bean>
            <!--配置另外一个 Bean-->
            <bean id="英语" class=" com.ssm..English"></bean>
        </beans>
```

（3）BeanFactory 采用了工厂设计模式，即 Bean 容器模式，负责读取 Bean 的配置文件，管理对象的生成、加载，维护 Bean 对象与 Bean 对象之间的依赖关系，负责 Bean 的生命周期。对于简单的应用程序来说，使用 BeanFactory 就已经足够管理 Bean 了，在对象的管理上可以获得许多便利性。

org.springframework.beans.factory.BeanFactory 是一个顶级接口，包含管理 Bean 的各种方法。Spring 框架也提供了一些实现该接口的类。

org.springframework.beans.factory.xml.XmlBeanFactory 是 BeanFactory 常用的实现类，根据配置文件中的定义装载 Bean。要创建 XmlBeanFactory，需要传递一个 FileInputStream 对象，该对象把 XML 文件提供给工厂。代码可以写成：

```
BeanFactory factory=new XmlBeanFactory( new FileInputStream("applicationContext.xml "));
```

BeanFactory 的常用方法如下：

- getBean(String name)：可根据 Bean 的 id 生成该 Bean 的对象。
- getBean(String name,Class requiredType)：可根据 Bean 的 id 和相应类生成该 Bean 的对象。

（4）ApplicationContext 接口提供高级功能的容器，基本功能与 BeanFactory 很相似，但它还有以下功能：

- 提供访问资源文件更方便的方法。
- 支持国际化消息。
- 提供文字消息解析的方法。
- 可以发布事件，对事件感兴趣的 Bean 可以接收到这些事件。

ApplicationContext 接口的常用实现类有以下 3 个。

- FileSystemXmlApplicationContext：从文件系统中的 XML 文件加载上下文中定义的信息。
- ClassPathXmlApplicationContext：从类路径中的 XML 文件加载上下文中定义的信息，把上下文定义的文件当成类路径资源。
- XmlWebApplicationContext：从 Web 系统中的 XML 文件加载上下文中定义的信息。

其中，FileSystemXmlApplicationContext 和 ClassPathXmlApplicationContext 的代码编写如下：

```
ApplicationContext context=new FileSystemXmlApplicationContext("d:/applicationContext.xml
 ");
ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml ");
```

第 1 种使用文件系统的方式来查询配置文件，此时 applicationContext.xml 文件位于 D 盘下。第 2 种使用类路径来查询配置文件，此时 applicationContext.xml 文件位于项目的 src 目录底下。

FileSystemXmlApplicationContext 和 ClassPathXmlApplicationContext 的区别是：

FileSystemXmlApplicationContext 只能在指定的路径中查询 applicationContext.xml 配置文件，而 ClassPathXmlApplicationContext 可以在整个类路径中查询 applicationContext.xml。

2. IoC/DI 实现方式

如前所述，依赖注入（DI）和控制反转（IoC）是对同一件事情的不同描述。依赖注入的作用是在使用 Spring 框架创建对象时，动态地将其所依赖的对象注入 Bean 组件中，其实现方式通常有两种：一种是属性 setter()方法注入；另一种是构造方法注入。具体介绍如下：

- 属性 setter()方法注入：IoC 容器使用 setter()方法注入被依赖的实例。通过调用无参构造器或无参静态工厂方法实例化 Bean 后，调用该 Bean 的 setter()方法，即可实现基于 setter()方法的依赖注入。该方式简单、直观，而且容易理解，所以 Spring 的设置注入被大量使用。
- 构造方法注入：IoC 容器使用构造方法注入被依赖的实例。基于构造方法的依赖注入通过调用带参数的构造方法来实现，每个参数代表着一个依赖。

【示例 1-1】下面以常用的 setter()方法注入的方式为例讲解 Spring 容器在应用中是如何实现依赖注入的。

（1）在 Eclipse 集成开发环境中创建一个名为 chapter01 的动态 Web 项目，将 Spring 的 4 个基础包以及 commons-logging 的 JAR 包复制到 lib 目录中，并发布到类路径下，如图 1.4 所示。

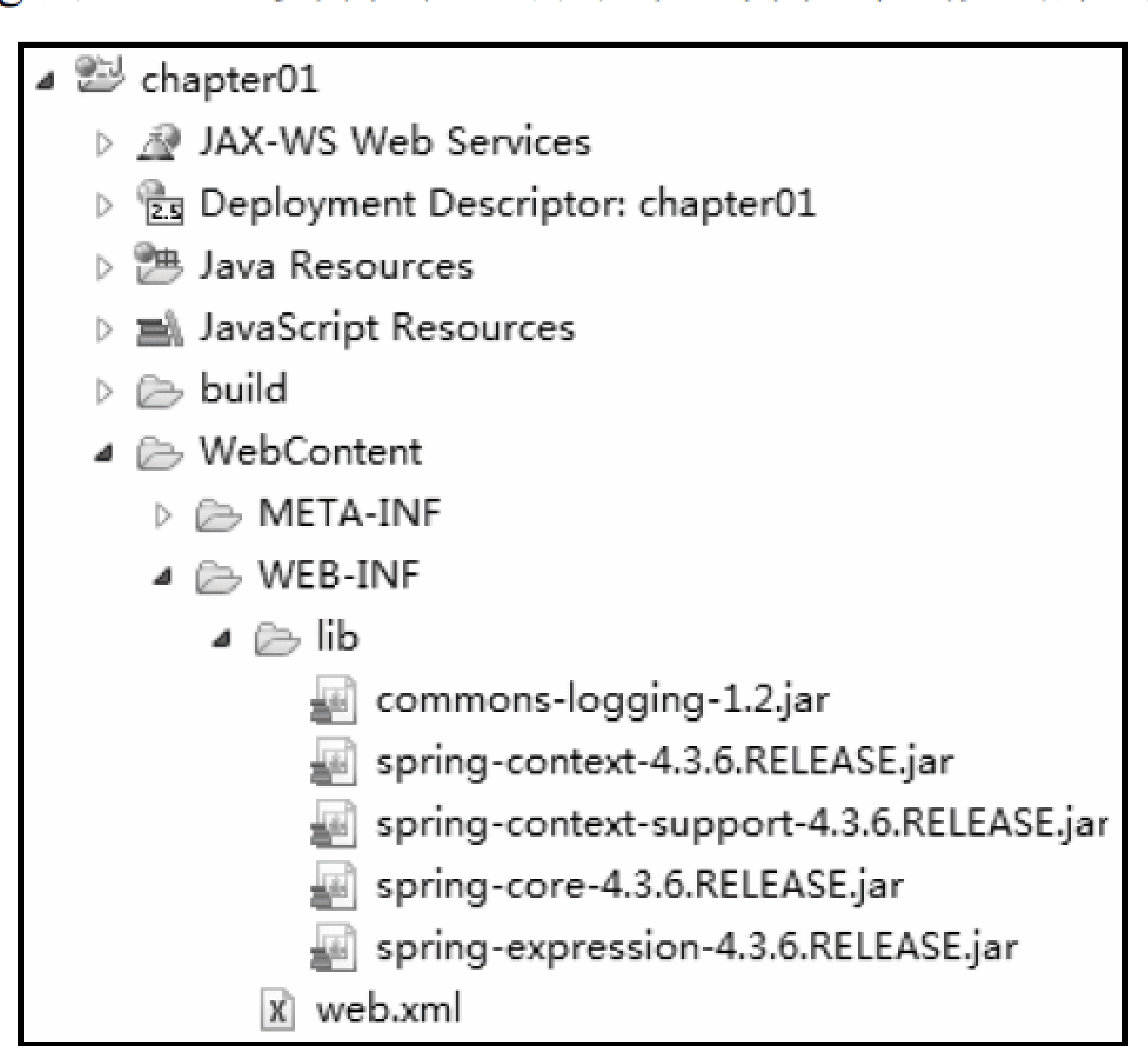


图 1.4　创建项目导入包

（2）在 src 目录下创建一个 com.ssm.ioc_di 包，并在包中创建接口 UserDao，然后在接口中定义一个 login()方法，如文件 1.1 所示。

文件 1.1　UserDao.java

```
package com.ssm.ioc_di;
public interface UserDao {
// 定义 login()方法
    public void login();
}
```

（3）在 com.ssm.ioc_di 包中创建 UserDao 接口的实现类 UserDaoImpl，该类需要实现接口中的 login()方法，并在方法中编写一条输出语句，如文件 1.2 所示。

文件 1.2　UserDaoImpl.java

```
package com.ssm.ioc_di;
public class UserDaoImpl implements UserDao {
// 实现 login()方法
    public void login() {
     System.out.println("UserDao login");
    }
}
```

（4）在 src 目录下创建 Spring 的配置文件 applicationContext.xml，并在配置文件中创建一个 id 为 UserDao 的 Bean，如文件 1.3 所示。

文件 1.3　applicationContext.xml

```
01     <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
03        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04        xsi:schemaLocation="http://www.springframework.org/schema/beans
05                http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
06      <!-- 将指定类配置给 Spring，让 Spring 创建其对象的实例 -->
07      <bean id="UserDao" class="com.ssm.ioc_di.UserDaoImpl" />
08     </beans>
```

在文件 1.3 中，第 01~05 代码中包含一些约束信息，其中“spring-beans-4.3.xsd”与 Spring 的版本 4.3 相对应；第 07 行代码表示在 Spring 容器中创建一个 id 为 UserDao 的 Bean 实例，其中 class 属性用于指定需求实例化 Bean 的类。

注　意
Spring 配置文件的名称可以自定义，通常默认命名为 applicationContext.xml。

（5）在 com.ssm.ioc_di 包中创建测试类 IoC，并在类中编写 main()方法及实现 IoC 的代码，如文件 1.4 所示。

文件 1.4　IoC.java

```
package com.ssm.ioc_di;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class IoC {
    public static void main(String[] args) {
     //1.初始化 Spring 容器，加载配置文件
     ApplicationContext applicationContext=
                    new ClassPathXmlApplicationContext("applicationContext.xml");
     //2.通过容器获取 userDao 实例
     UserDao userDao=(UserDao)applicationContext.getBean("userDao");
```

```
        //3.调用实例中的 login()方法
        userDao.login();
    }
}
```

程序执行后，控制台输出结果如图 1.5 所示。从中可以看出，控制台成功输出了 UserDaoImpl 类中的输出语句。在文件 1.4 的 main()方法中，并没有通过 new 关键字来创建 UserDao 接口的实现类对象，而是通过 Spring 容器来获取的实现类对象，这就是 Spring IoC（控制反转）的工作机制。

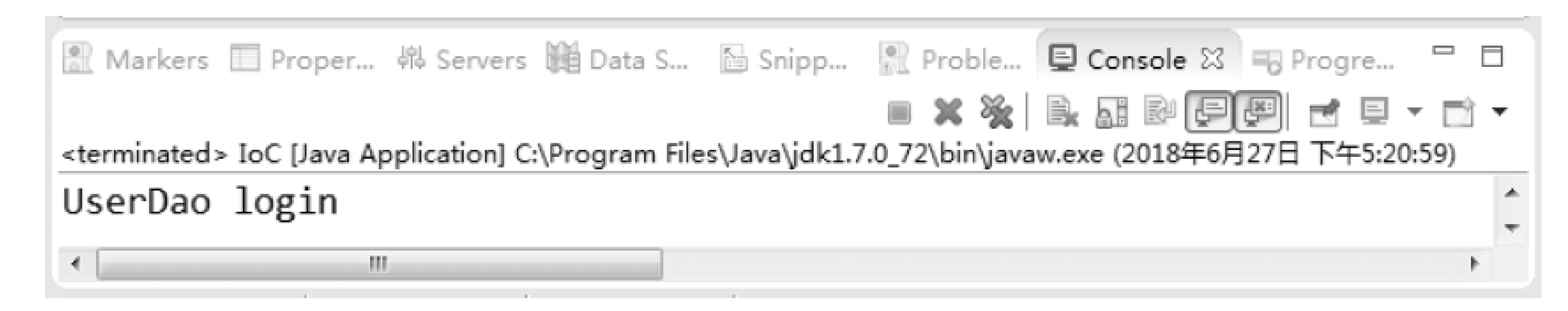


图 1.5　运行结果

（6）在 com.ssm.ioc_di 包中创建接口 UserService，并编写一个 login()方法，如文件 1.5 所示。

文件 1.5　UserService.java

```
package com.ssm.ioc_di;
public interface UserService {
    public void login();
}
```

（7）在 com.ssm.ioc_di 包中创建接口 UserService 的实现类 UserServiceImpl，在类中声明 userDao 属性，并添加属性的 setter()方法；同时编写 login()方法。具体代码如文件 1.6 所示。

文件 1.6　UserServiceImpl.java

```
package com.ssm.ioc_di;
public class UserServiceImpl implements UserService {
    //声明 userDao 属性
    private UserDao userDao;
    //添加 userDao 属性的 setter()方法，用于实现依赖注入
    public void setUserDao(UserDao userDao) {
     this.userDao = userDao;
    }
    //实现接口中的方法
    public void login() {
     // 调用 userDao 中的 login()方法，并执行输出语句
     this.userDao.login();
     System.out.println("userService login");
    }
}
```

（8）在配置文件 applicationContext.xml 中创建一个 id 为 userService 的 Bean，该 Bean 用于实例化 UserServiceImpl 类的信息，并将 name 为 userDao 的实例注入 userService 中，其代码如下所示。

```
<!-- 添加一个 id 为 userService 的 Bean -->
```

```
<bean id="userService" class="com.ssm.ioc_di.userServiceImpl">
    <!-- 将 name 为 userDao 的 Bean 实例注入 userService 实例中 -->
    <property name="userDao" ref="userDao"/>
</bean>
```

在上述代码中，<property>是<bean>元素的子元素，用于调用 Bean 实例中的 setUserDao()方法完成属性赋值，从而实现依赖注入。其 name 属性表示 Bean 实例中的相应属性名，ref 属性用于指定其属性值。

（9）在 com.ssm.ioc_di 包中创建测试类 DI，如文件 1.7 所示。

文件 1.7 DI.java

```
package com.ssm.ioc_di;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DI {
    public static void main(String[] args) {
     // 1.初始化 Spring 容器，加载配置文件
     ApplicationContext applicationContext =
                    new ClassPathXmlApplicationContext("applicationContext.xml");
     // 2.通过容器获取 userService 实例
     UserService userService = (UserService) applicationContext.getBean("userService");
     // 3.调用实例中的 login()方法
     userService.login();
    }
}
```

此时运行效果如图 1.6 所示。从中可以看出，使用 Spring 容器通过 UserService 实现类中的 login()方法调用了 UserDao 实现类中的 login()方法，并输出了结果。这就是 Spring 容器属性 setter 注入的方式，也是实际开发中常用的一种方式。

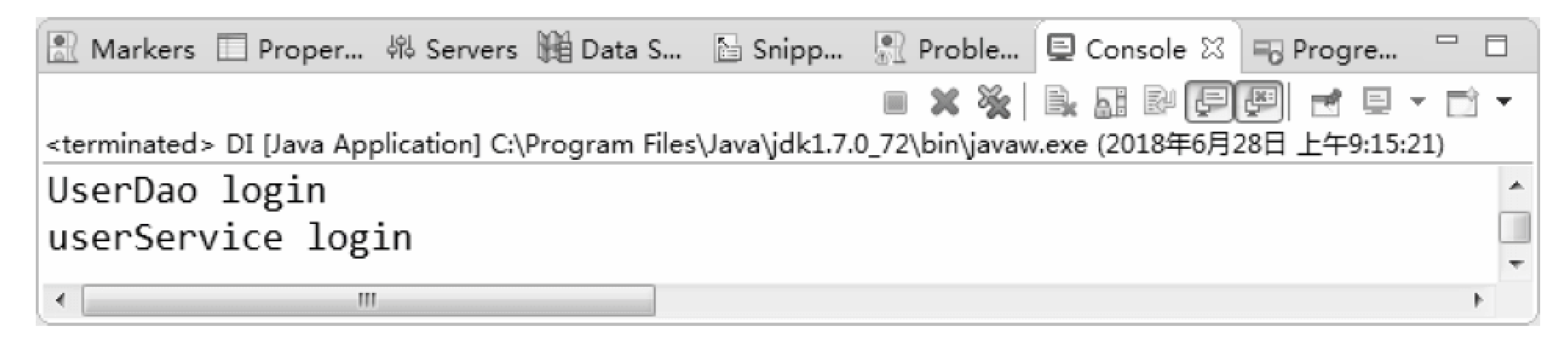


图 1.6 运行结果

为了方便读者理解，图 1.7 给出整个项目目录结构。

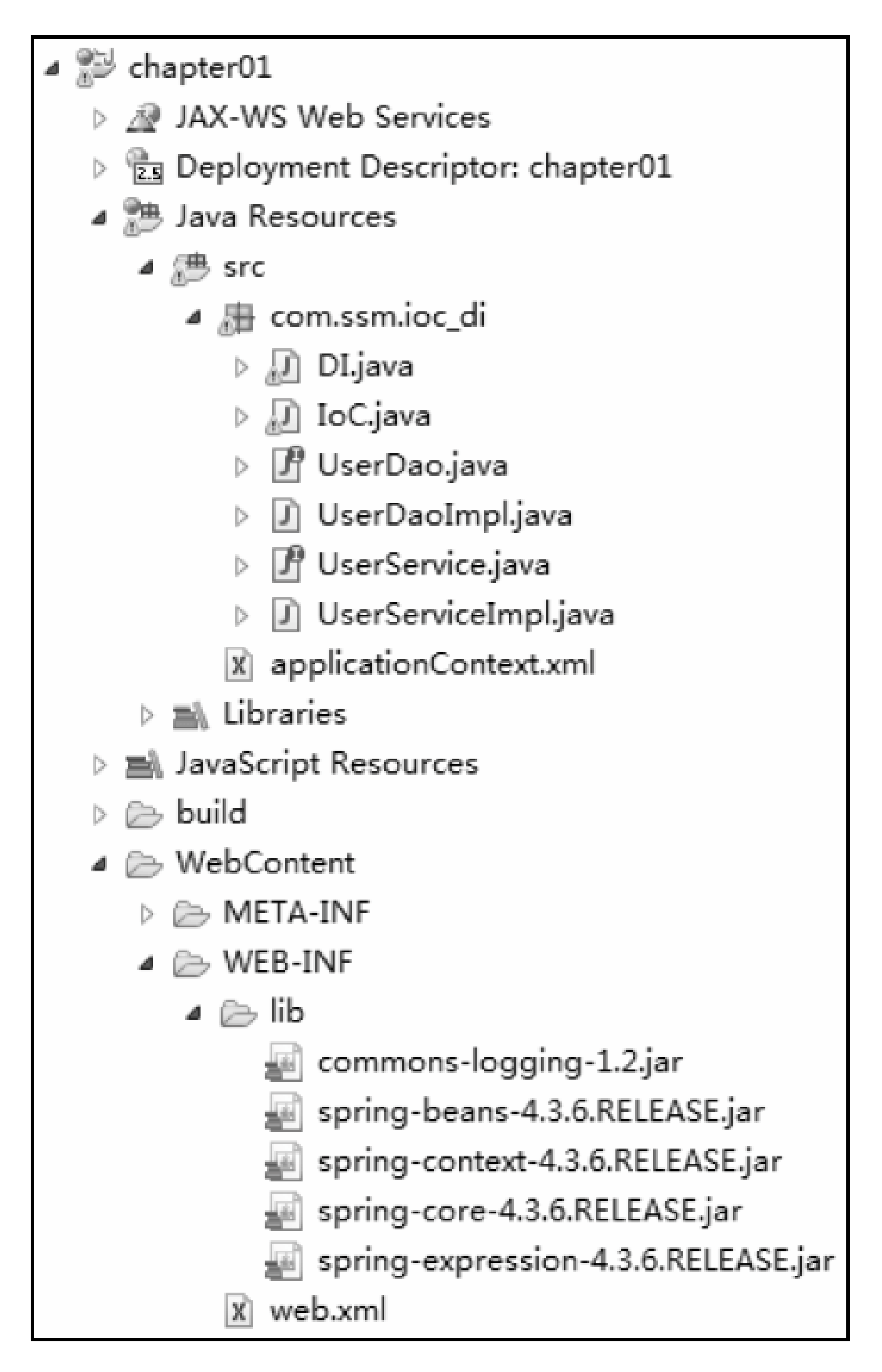


图 1.7　项目目录结构

1.3　习　　题

1．简述什么是 Spring 的 IoC 和 DI。

2．上机练习 Spring 的 IoC 和 DI 的实现。

第 2 章

Spring 中的 Bean

第 1 章讲解了 Spring 的控制反转/依赖注入并演示了如何实现它。本章将在此基础上针对 Spring 中 Bean 的相关知识进行讲解。

本章主要涉及的知识点如下：

- Bean 的配置：了解 Bean 的常用属性及其子元素。
- Bean 的作用域：了解 Bean 的作用域种类及常用作用域 singleton 和 prototype。
- Bean 的装配方式：了解 Bean 的 3 种装配方式。

2.1 Bean 的配置

Spring 如同一个工厂，用于生产和管理 Spring 容器中的 Bean。要使用这个工厂，需要开发者对 Spring 的配置文件进行配置。在实际开发中，最常采用 XML 格式的配置方式，即通过 XML 文件来注册并管理 Bean 之间的依赖关系。本节将使用 XML 文件的形式对 Bean 的属性和定义进行讲解。

在 Spring 中，XML 配置文件的根元素是<beans>，<beans>中可以包含多个<bean>子元素，每一个<bean>子元素定义了一个 Bean，并描述了该 Bean 如何被装配到 Spring 容器中。<bean>子元素中包含多个属性和子元素，常用的属性和子元素如表 2.1 所示。

表 2.1　<bean>元素的常用属性和子元素

属性或子元素名称	说明
id	Bean 的唯一标识符，Spring 容器对 Bean 的配置、管理都通过该属性进行
name	Spring 容器通过此属性进行配置和管理，name 属性可以为 Bean 指定多个名称，每个名称之间用逗号或分号隔开

（续表）

属性或子元素名称	说明
class	指定 Bean 的实现类，它必须使用类的全限定名
scope	用于设定 Bean 实例的作用域，其属性值有 singleton（单例）、prototype（原型）、request、session、global Session、application 和 websocket，默认值为 singleton
constructor-arg	<bean>元素的子元素，可以使用此元素传入构造参数进行实例化。该元素的 index 属性指定构造参数的序号（从 0 开始），type 属性指定构造参数的类型，参数值可以通过 ref 属性或 value 属性直接指定，也可以通过 ref 或 value 子元素指定
property	<bean>元素的子元素，用于调用 Bean 实例中的 setter()方法完成属性赋值，从而完成依赖注入。该元素的 name 属性指定 Bean 实例中的相应属性名，ref 属性或 value 属性用于指定参数值
ref	<constructor-arg>、<property>等元素的属性或子元素，可以用于指定对 Bean 工厂中某个 Bean 实例的引用
value	<constructor-arg>、<property>等元素的属性或子元素，可以用于直接给定一个常量值
list	用于封装 List 或数组属性的依赖注入
set	用于封装 Set 类型属性的依赖注入
map	用于封装 Map 类型属性的依赖注入
entry	<map>元素的子元素，用于设置一个键值对。其 key 属性指定字符串类型的键值，ref 属性或 value 属性直接指定其值，也可以通过 ref 或 value 子元素指定其值

表 2.1 中只介绍了<bean>元素的常用属性和子元素，实际上<bean>元素还有很多属性和子元素，读者可以到网上查阅相关资料进行获取。

在 Spring 的配置文件中，通常一个普通的 Bean 只需要定义 id（或 name）和 class 两个属性即可。定义 Bean 的方式如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!-- 将指定类配置给 Spring，让 Spring 创建其对象的实例  -->
    <!—使用 id 属性定义 bean1，其对应的实现类为 com.ssm.Bean1  -->
        <bean id="bean1" class="com.ssm.Bean1" />
    <!--使用 name 属性定义 bean2，其对应的实现类为 com.ssm.Bean2  -->
    <bean name="bean2" class="com.ssm.Bean2" />
</beans>
```

在上述代码中，分别使用 id 属性和 name 属性定义了两个 Bean，并使用 class 元素指定其对应的实现类。

注　意
如果在 Bean 中未指定 id 和 name，那么 Spring 会将 class 值当作 id 使用。

2.2　Bean 的作用域

通过 Spring 容器创建一个 Bean 的实例时，不仅可以完成 Bean 的实例化，还可以为 Bean 指定特定的作用域。本节将主要讲解 Bean 的作用域相关的知识。

2.2.1　作用域的种类

Spring 4.3 中为 Bean 的实例定义了 7 种作用域，如表 2.2 所示。其中，singleton 和 prototype 是常用的两种，在接下来的两小节中将会对这两种作用域进行详细讲解。

表 2.2　Bean 的作用域

作用域名称	说明
singleton（单例）	使用 singleton 定义的 Bean 在 Spring 容器中将只有一个实例，也就是说，无论有多少个 Bean 引用它，始终将指向同一个对象，这也是 Spring 容器默认的作用域
prototype（原型）	每次通过 Spring 容器获取 prototype 定义的 Bean 时，容器都将创建一个新的 Bean 实例
request	在一次 HTTP 请求中，容器会返回该 Bean 的同一个实例，对不同的 HTTP 请求则会产生一个新的 Bean，而且该 Bean 仅在当前 HTTP Request 内有效
session	在一次 HTTP Session 中，容器会返回该 Bean 的同一个实例，对不同的 HTTP 请求则会产生一个新的 Bean，而且该 Bean 仅在当前 HTTP Session 内有效
globalSession	在一个全局的 HTTP Session 中，容器会返回该 Bean 的同一个实例，仅在使用 portlet 上下文时有效
application	为每个 ServletContext 对象创建一个实例，仅在 Web 相关的 ApplicationContext 中有效
websocket	为每个 websocket 对象创建一个实例，仅在 Web 相关的 ApplicationContext 中生效

2.2.2　singleton 作用域

singleton 是 Spring 容器默认的作用域，当 Bean 的作用域为 singleton 时，Spring 容器就只会存在一个共享的 Bean 实例，并且所有对 Bean 的请求，只要 id 与该 Bean 的 id 属性相匹配，就会返回同一个 Bean 的实例。singleton 作用域对于无会话状态的 Bean（如 Dao 组件、Service 组件）来说是最理想的选择。

在 Spring 配置文件中，Bean 的作用域是通过<bean>元素的 scope 属性来指定的，该属性值可以设置为 singleton、prototype、request、session、globalSession、application、websocket 七个值，分别表示表 2.2 中的 7 种作用域。要将作用域定义成 singleton，需将 scope 的属性值设置为 singleton，其示例代码如下。

```
<bean id="scope"  class="com.ssm.scope.Scope"  scope="singleton" />
```

【示例 2-1】下面通过一个案例来进一步演示 singleton 作用域。

（1）在 Eclipse 中创建一个名为 chapter02 的 Web 项目，在该项目的 lib 目录中加入 Spring 支

持和依赖的 JAR 包（在第 1 章相关内容基础上增加 spring-aop-4.3.6.RELEASE.jar 依赖包，并发布到类路径下）。

（2）在 chapter02 项目的 src 目录下创建一个 com.ssm.scope 包，在该包中创建 Scope 类，该类不需要写什么方法，如文件 2.1 所示。

文件 2.1　Scope.java

```
package com.ssm.scope;
public class Scope {
}
```

（3）在 com.ssm.scope 包中创建 Spring 的配置文件 applicationContext.xml，并在配置文件中创建一个 id 为 scope 的 Bean，通过 class 属性指定其对应的实现类为 Scope，如文件 2.2 所示。

文件 2.2　applicationContext.xml

```
    <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!-- 将指定类配置给 Spring，让 Spring 创建其对象的实例 -->
    <bean id="scope" class="com.ssm.scope.Scope" />
    </beans>
```

（4）在 com.ssm.scope 包中创建测试类 ScopeTest 来测试 singleton 作用域，如文件 2.3 所示。

文件 2.3　ScopeTest.java

```
package com.ssm.scope;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class ScopeTest {
     public static void main(String[] args) {
    // 1.初始化 Spring 容器，加载配置文件
    ApplicationContext applicationContext =
                   new ClassPathXmlApplicationContext("applicationContext.xml");
    // 2.输出获得的实例
    System.out.println(applicationContext.getBean("scope"));
    System.out.println(applicationContext.getBean("scope"));
      }
}
```

执行程序后，控制台的输出结果如图 2.1 所示。从中可以看出，两次输出的结果相同，这说明 Spring 容器只创建了一个 Scope 类的实例。

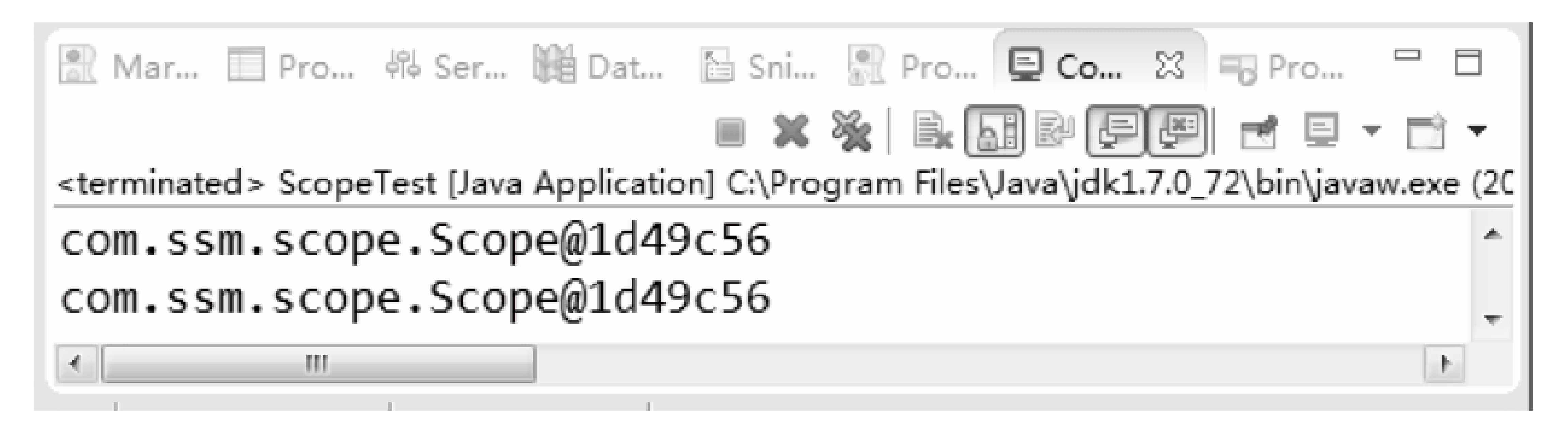


图 2.1 运行结果

注 意

如果不设置 scope="singleton"，其输出结果也是一个实例，因为 Spring 容器默认的作用域就是 singleton。

2.2.3 prototype 作用域

对需要保持会话状态的 Bean 应用使用 prototype 作用域。在使用 prototype 作用域时，Spring 容器会为每个对该 Bean 的请求都创建一个新的实例。

要将 Bean 定义为 prototype 作用域，只需在配置文件中将<bean>元素的 scope 属性值设置为 prototype 即可，其示例代码如下。

```
<bean id="scope" class="com.ssm.scope.Scope" scope="prototype"/>
```

将 2.2.2 小节中的配置文件更改成上述代码形式后，再次运行测试类 ScopeTest，控制台的输出结果如图 2.2 所示。从中可以看到，两次输出的 Bean 实例并不相同，这说明在 prototype 作用域下创建了两个不同的 Scope 实例。

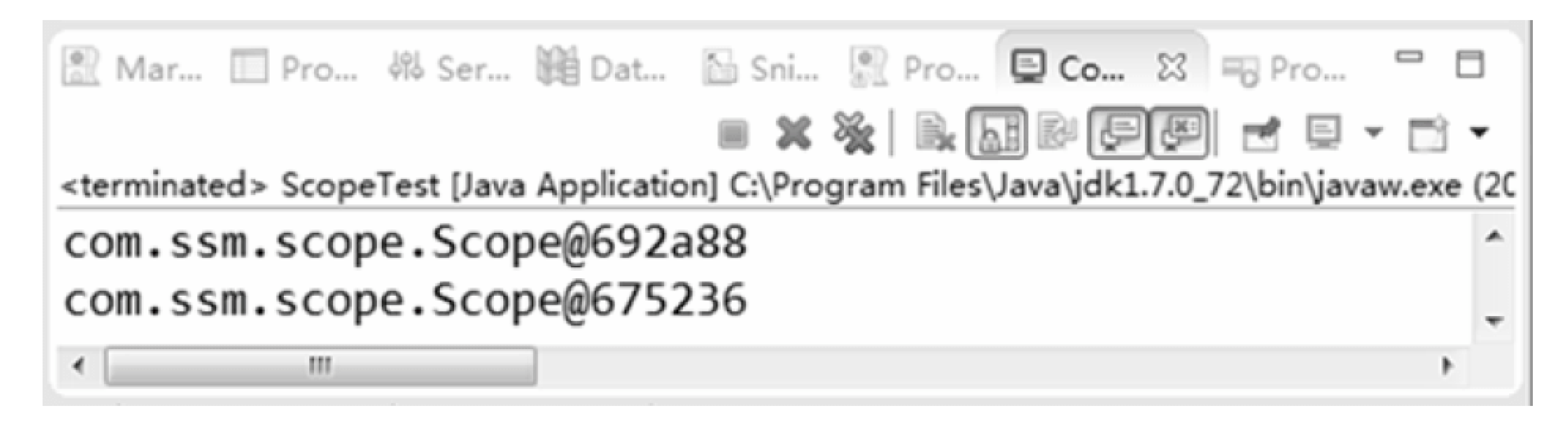


图 2.2 运行结果

2.3 Bean 的装配方式

Bean 的装配可以理解为依赖关系注入，Bean 的装配方式即 Bean 依赖注入的方式。Spring 容器支持多种形式的 Bean 装配方式，如基于 XML 的装配、基于 Annotation（注解）的装配和自动装配等。本节主要讲解这 3 种装配方式的使用。

2.3.1　基于 XML 的装配

Spring 提供了两种基于 XML 的装配方式：设值注入（Setter Injection）和构造注入（Constructor Injection）。下面讲解如何在 XML 配置文件中使用这两种注入方式来实现基于 XML 的装配。

在 Spring 实例化 Bean 的过程中，Spring 首先会调用 Bean 的默认构造方法来实例化 Bean 对象，然后通过反射的方式调用 setter()方法来注入属性值。因此，设值注入要求一个 Bean 必须满足以下两点要求：

- Bean 类必须提供一个默认的无参构造方法。
- Bean 类必须为需要注入的属性提供对应的 setter()方法。

使用设值注入时，在 Spring 配置文件中需要使用<bean>元素的子元素<property>来为每个属性注入值；而使用构造注入时，在配置文件中需要使用<bean>元素的子元素<constructor-arg>来定义构造方法的参数，可以使用其 value 属性（或子元素）来设置该参数的值。

【示例 2-2】下面通过一个案例来演示基于 XML 方式的 Bean 的装配。

（1）在项目 chapter02 的 src 目录下创建一个 com.ssm.assemble 包，在该包中创建 User 类，并在类中定义 userName、password 和 list 集合 3 个属性及对应的 setter()方法，如文件 2.4 所示。

文件 2.4　User.java

```
package com.ssm.assemble;
import java.util.List;
public class User {
    private String userName;
    private String password;
    private List<String> list;
     /**
      * 1.使用构造注入
      * 1.1 提供带所有参数的构造方法
     */
     public User(String userName, String password, List<String> list) {
     super();
     this.userName = userName;
     this.password = password;
     this.list = list;
     }
     @Override
     public String toString() {
     return "User [userName=" + userName + ", password=" + password + ", list=" + list + "]";
     }
     /**
      * 2.使用设值注入
      * 2.1 提供默认空参构造方法
      * 2.2 为所有属性提供 setter()方法
     */
```

```
26        public User() {
27        super();
28        }
29        public void setUserName(String userName) {
30        this.userName = userName;
31        }
32        public void setPassword(String password) {
33        this.password = password;
34        }
35        public void setList(List<String> list) {
36        this.list = list;
37        }
38    }
```

在文件 2.4 的第 11~16 行、第 26~28 行，由于要使用构造注入，因此需要编写有参和无参的构造方法。

（2）在 Spring 的配置文件 application.xml（文件 2.2）中，增加通过构造注入和设值注入的方法装配 User 实例的两个 Bean，代码如下所示。

```
<bean id="user1" class="com.ssm.assemble.User">
    <constructor-arg index="0" value="zhangsan" />
    <constructor-arg index="1" value="111111" />
    <constructor-arg index="2">
        <list>
            <value>"constructorValue1"</value>
            <value>"constructorValue2"</value>
        </list>
    </constructor-arg>
</bean>
<bean id="user2" class="com.ssm.assemble.User">
    <property name="userName" value="lisi"></property>
    <property name="password" value="222222"></property>
    <property name="list">
        <list>
            <value>"listValue1"</value>
            <value>"listValue2"</value>
        </list>
    </property>
</bean>
```

在上述代码中，<constructor-arg>元素用于定义构造方法的参数，其属性 index 表示其索引（从 0 开始），value 属性用于设置注入的值，其子元素<list>为 User 类中对应的 list 集合属性注入值。然后又使用设值注入方法装配 User 类的实例，其中<property>元素用于调用 Bean 实例中的 setter()方法完成属性赋值，从而完成依赖注入，而其子元素<list>同样为 User 类中对应的 list 集合属性注入值。

（3）在 com.ssm.assemble 包中创建测试类 XmlAssembleTest，在类中分别获取并输出配置文件

中的 user1 和 user2 实例，如文件 2.5 所示。

文件 2.5　XmlAssembleTest.java

```
package com.ssm.assemble;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class XmlAssembleTest {
     public static void main(String[] args) {
    // 1.初始化 Spring 容器，加载配置文件
    ApplicationContext applicationContext
                  = new ClassPathXmlApplicationContext("applicationContext.xml");
    // 2.输出获得的实例
    System.out.println(applicationContext.getBean("user1"));
    System.out.println(applicationContext.getBean("user2"));
     }
}
```

执行程序后，控制台输出结果如图 2.3 所示。可以看出，已经成功地使用基于 XML 装配的构造注入和设值注入两种方式装配了 User 实例。

```
<terminated> XmlAssembleTest [Java Application] C:\Program Files\Java\jdk1.7.0_72\bin\javaw.exe (2018年7月13日 下午5:43:29)
User [userName=zhangsan, password=111111, list=["constructorValue1", "constructorValue2"]]
User [userName=lisi, password=222222, list=["listValue1", "listValue2"]]
```

图 2.3　运行结果

2.3.2　基于 Annotation 的装配

在 Spring 中，尽管使用 XML 配置文件可以实现 Bean 的装配工作，但如果应用中有很多 Bean，就会导致 XML 配置文件过于臃肿，给以后的维护和升级工作带来一定的困难。为此，Spring 提供了对 Annotation（注解）技术的全面支持。

Spring 中定义了一系列的注解，常用的注解如表 2.3 所示。

表 2.3　Spring 的常用注解

注解名称	说明
@Component	可以使用此注解描述 Spring 中的 Bean，但它是一个泛化的概念，仅仅表示一个组件（Bean），并且可以作用在任何层次。使用时只需将该注解标注在相应类上即可
@Repository	用于将数据访问层（DAO 层）的类标识为 Spring 中的 Bean，其功能与@Component 相同
@Service	通常作用在业务层（Service 层），用于将业务层的类标识为 Spring 中的 Bean，其功能与@Component 相同
@Controller	通常作用在控制层（如 Spring MVC 的 Controller），用于将控制层的类标识为 Spring 中的 Bean，其功能与@Component 相同

（续表）

注解名称	说明
@Autowired	用于对 Bean 的属性变量、属性的 setter()方法及构造方法进行标注，配合对应的注解处理器完成 Bean 的自动配置工作。默认按照 Bean 的类型进行装配
@Resource	其作用与@Autowired 一样，区别在于@Autowired 默认按 Bean 类型装配，而@Resource 默认按照 Bean 实例名称进行装配。@Resource 中有两个重要属性：name 和 type。Spring 将 name 属性解析为 Bean 实例名称，type 属性解析为 Bean 实例类型。若指定 name 属性，则按实例名称进行装配；若指定 type 属性，则按 Bean 类型进行装配；若都不指定，则先按 Bean 实例名称装配，不能匹配时再按照 Bean 类型进行装配；若都无法匹配，则抛出 NoSuchBeanDefinitionException 异常
@Qualifier	与@Autowired 注解配合使用，会将默认的按 Bean 类型装配修改为按 Bean 的实例名称装配，Bean 的实例名称由@Qualifier 注解的参数指定

注　意

在表 2.3 的几个注解中，虽然@Repository、@Service 和@Controller 的功能与@Component 注解的功能相同，但为了使标注类本身用途更加清晰，建议在实际开发中使用@Repository、@Service 和@Controller 分别对实现类进行标注。

【示例 2-3】接下来，通过一个案例来演示如何通过这些注解来装配 Bean。

（1）在 chapter02 项目的 src 目录下创建一个 com.ssm.annotation 包，在该包中创建接口 UserDao，并在接口中定义一个 save()方法，如文件 2.6 所示。

文件 2.6　UserDao.java

```
package com.ssm.annotation;
public interface UserDao {
     public void save();
}
```

（2）在 com.ssm.annotation 包中创建 UserDao 接口的实现类 UserDaoImpl，该类需要实现接口中的 save()方法，如文件 2.7 所示。

文件 2.7　UserDaoImpl.java

```
package com.ssm.annotation;
import org.springframework.stereotype.Repository;
// 使用@Repository 注解将 UserDaoImpl 类标识为 Spring 中的 Bean
@Repository("userDao")
public class UserDaoImpl implements UserDao {
    public void save() {
     System. out. println("userDao.save()");
    }
}
```

在文件 2.7 中，首先使用@Repository 注解将 UserDaoImpl 类标识为 Spring 中的 Bean，其写法相当于配置文件中<bean id="user Dao" class=" com.ssm.annotation.UserDaoImpl />的编写。然后在

save()方法中打印一句话，用于验证是否成功调用了该方法。

（3）在 com.ssm.annotation 包中创建接口 UserService，在接口中同样定义一个 save()方法，如文件 2.8 所示。

文件 2.8　UserService.java

```
package com.ssm.annotation;
public interface UserService {
    public void save();
}
```

（4）在 com.ssm.annotation 包中创建 UserService 接口的实现类 UserServiceImpl，该类需要实现接口中的 save()方法，如文件 2.9 所示。

文件 2.9　UserServiceImpl.java

```
package com.ssm.annotation;
import javax.annotation.Resource;
import org.springframework.stereotype.Service;
// 使用@Service 注解将 UserServiceImpl 类标识为 Spring 中的 Bean
@Service("userService")
public class UserServiceImpl implements UserService {
// 使用@Resource 注解注入
    @Resource(name="userDao")
    private UserDao userDao;
    public void save() {
     this.userDao. save();
     System.out.println("执行 userService.save()");
    }
}
```

在文件 2.9 中，首先使用@Service 注解将 UserServiceImpl 类标识为 Spring 中的 Bean，这相当于配置文件中<bean id= "userService" class="com.ssm.annotation.UserServiceImpl"/>的编写；然后使用@Resource 注解标注在属性 userDao 上，这相当于配置文件中<property name="userDao" ref=" userDao"/>的写法；最后在该类的 save()方法中调用 userData 中的 save()方法，并输出一句话。

（5）在 com.ssm.annotation 包中创建控制器类 UserController，如文件 2.10 所示。

文件 2.10　UserController.java

```
package com.ssm.annotation;
import javax.annotation.Resource;
import org.springframework.stereotype.Controller;
// 使用@Controller 注解将 UserController 类标识为 Spring 中的 Bean
@Controller("UserController")
public class UserController {
// 使用@Resource 注解注入
    @Resource(name="userService")
    private UserService userService;
    public void save(){
```

```
        this.userService.save();
        System.out.println("运行 userController.save()");
    }
}
```

首先使用@Controller 注解标注了 UserController 类，这相当于在配置文件中编写<bean id=" userController" class=" com.ssm.annotation.UserController"/>；然后使用@Resource 注解标注在 userService 属性上，这相当于在配置文件中编写<property name="userService" ref=" userService"/>；最后在其 save()方法中调用了 userService 中的 save()方法，并输出一句话。

（6）在 com.ssm.annotation 包中创建配置文件 beans1.xml，在配置文件中编写基于 Annotation 装配的代码，如文件 2.11 所示。

文件 2.11　beans1.xml

```
01  <?xml version="1.0" encoding="UTF-8"?>
02  <beans xmlns="http://www.springframework.org/schema/beans"
03    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04    xmlns:context="http://www.springframework.org/schema/context"
05    xsi:schemaLocation="http://www.springframework.org/schema/beans
06    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
07    http://www.springframework.org/schema/context
08    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
09    <!-- 使用 context 命名空间在配置文件中开启相应的注解处理器 -->
10    <context:annotation-config />
11    <!-- 分别定义 3 个 Bean 实例 -->
12    <bean id="userDao" class="com.ssm.annotation.UserDaoImpl" />
13    <bean id="userService" class="com.ssm.annotation.UserServiceImpl" />
14    <bean id="userController" class="com.ssm.annotation.UserController" />
15  </beans>
```

从上述代码可以看出，文件 2.11 与之前的配置文件有很大不同。首先，在<beans>元素中增加了 04、07 和 08 行中包含 context 的约束信息；然后通过配置<context: annotation-config/>来开启注解处理器；最后分别定义了 3 个 Bean 对应的 3 个实例。与 XML 配置方式有所不同的是，这里不再需要配置子元素<property>。

上述 Spring 配置文件中的注解方式虽然较大程度地简化了 XML 文件中 Bean 的配置，但仍需在 Spring 配置文件中一一配置相应的 Bean，为此 Spring 注解提供了另一种高效的注解配置方式（对包路径下的所有 Bean 文件进行扫描），其配置方式如下：

```
<context: component- scan base-package="Bean 所在的包路径"/>
```

所以可以将上述文件 2.11 中的 09~14 行代码进行如下替换：

```
<!--使用 context 命名空间通知 Spring 扫描指定包下所有 Bean 类，进行注解解析-->
<context: component-scan base-package="com.ssm.annotation"/>
```

注　意

Spring 4.0 以上版本使用上面的代码对指定包中的注解进行扫描前，需要先向项目中导入 SpringAOP 的 JAR 包 spring-aop-4.3.6.RELEASE.jar，否则程序在运行时会报出“java.lang.NoClassDefFoundError:org/springframework/aop/TargetSource”错误。

（7）在 com.ssm.annotation 包中创建测试类 AnnotationAssembleTest，在类中编写测试方法并定义配置文件的路径，然后通过 Spring 容器加载配置文件并获取 UserController 实例，最后调用实例中的 save()方法，如文件 2.12 所示。

文件 2.12　AnnotationAssembleTest.java

```java
package com.ssm.annotation;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class AnnotationAssembleTest {
    private static ApplicationContext applicationContext;
    public static void main(String[] args) {
     // 定义配置文件路径
     String xmlPath = "com/ssm/annotation/beans1.xml";
     applicationContext = new ClassPathXmlApplicationContext(xmlPath);
     // 获取 UserController 实例
     UserController userController =
                 (UserController) applicationContext.getBean("userController");
     // 调用 UserController 中的 save()方法
     userController.save();
    }
}
```

执行程序后，控制台的输出结果如图 2.4 所示。从中可以看到，Spring 容器已成功获取了 UserController 的实例，并通过调用实例中的方法执行了各层中的输出语句，这说明已成功实现了基于 Annotation 来装配 Bean 实例。

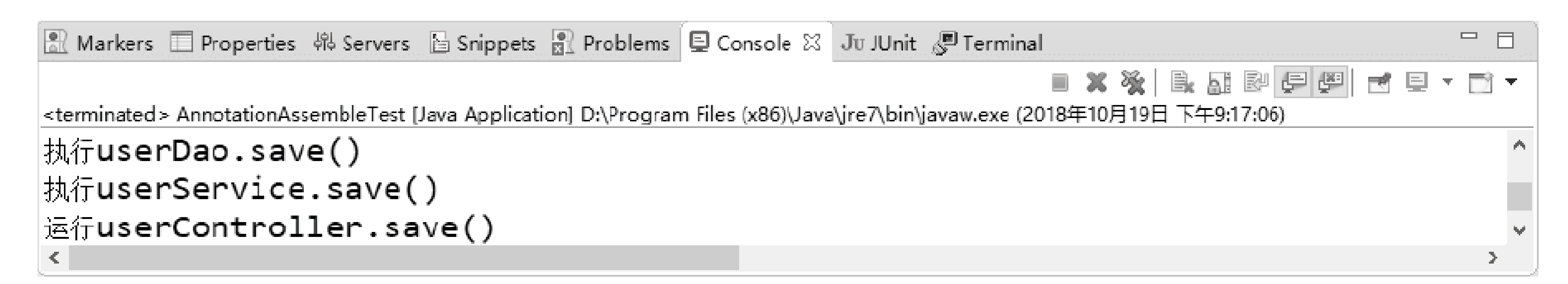


图 2.4　运行结果

注　意

上述案例中使用@Autowired 注解替换@Resource 注解也可以达到同样的效果。

2.3.3　自动装配

虽然使用注解的方式装配 Bean 在一定程度上减少了配置文件中的代码量，但是也有企业项目

中是没有使用注解方式开发的，那么有没有什么办法既可以减少代码量，又能够实现 Bean 的装配呢？答案是肯定的。Spring 的<bean>元素中包含一个 autowire 属性，我们可以通过设置 autowire 的属性值来自动装配 Bean。所谓自动装配，就是将一个 Bean 自动注入其他 Bean 的 Property 中。

autowire 属性有 5 个值，其值及说明如表 2.4 所示。

表 2.4 <bean>元素的 autowire 属性值及说明

属性值	说明
default（默认值）	由<bean>的上级标签<beans>的 default-autowire 属性值确定。例如 <beans default-autowire=" byName">，该<bean>元素中的 autowire 属性对应的属性值为 byName
byName	根据属性的名称自动装配。容器将根据名称查找与属性完全一致的 Bean，并将其属性自动装配
byType	根据属性的数据类型（Type）自动装配，如果一个 Bean 的数据类型兼容另一个 Bean 中属性的数据类型，则自动装配
constructor	根据构造函数参数的数据类型进行 byType 模式的自动装配
no	在默认情况下，不使用自动装配，Bean 依赖必须通过 ref 元素定义

【示例 2-4】下面通过修改 2.3.2 节中的案例来演示如何使用自动装配。

（1）修改 2.3.2 小节中的文件 2.9（UserServiceImpl.java）和文件 2.10（UserController.java），分别在这两个文件中增加类属性的 setter()方法。

（2）修改 2.3.2 小节中的配置文件 2.11（beans1.xml），将其修改成自动装配形式，如文件 2.13 所示。

文件 2.13 beans2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">
  <!-- 使用 bean 元素的 autowire 属性完成自动装配 -->
  <bean id="userDao" class="com.ssm.annotation.UserDaoImpl" />
  <bean id="userService"
       class="com.ssm.annotation.UserServiceImpl" autowire="byName" />
  <bean id="userController"
       class="com.ssm.annotation.UserController"  autowire="byName" />
</beans>
```

在上述配置文件中，用于配置 userService 和 userController 的<bean>元素中除了 id 和 class 属性外，还增加了 autowire 属性，并将其属性值设置为 byName。在默认情况下，配置文件中需要通过 ref 来装配 Bean，但设置了 autowire="byName"后，Spring 会自动寻找 userServiceBean 中的属性，并将其属性名称与配置文件中定义的 Bean 做匹配。由于 UserServiceImpl 中定义了 userDao 属性及其 setter()方法，这与配置文件中 id 为 userDao 的 Bean 相匹配，因此 Spring 会自动地将 id 为 userDao

的 Bean 装配到 id 为 userService 的 Bean 中。

执行程序后，控制台的输出结果与图 2.4 相同。

2.4 习　　题

1. 请简述 Bean 两种常用的作用域及其区别。
2. 请简述 Bean 几种装配方式的基本用法。

第 3 章

Spring AOP

Spring 的 AOP 模块是 Spring 框架体系结构中十分重要的内容，提供了面向切面编程的实现。本章将对 Spring AOP 的相关知识进行详细讲解。

本章主要涉及的知识点如下：

- AOP 概述：了解 AOP 的概念和作用，理解 AOP 中的相关术语。
- Aspect 开发：掌握基于 XML 的声明式 AspectJ 和基于注解的声明式 AspectJ。

3.1 Spring AOP 简介

本节主要介绍 AOP 的概念和作用，以及 AOP 中的相关术语，旨在让读者熟悉另一种编程方式，为后续学习打下基础。

3.1.1 什么是 AOP

AOP 的全称是 Aspect-Oriented Programming，即面向切面编程（也称面向方面编程），是面向对象编程（OOP）的一种补充，目前已成为一种比较成熟的编程方式。

在传统的业务处理代码中，通常都会进行事务处理、日志记录等操作。虽然使用 OOP 可以通过组合或者继承的方式来达到代码的重用，但如果要实现某个功能（如日志记录），相同的代码仍然会分散到各个方法中。这样，如果想要关闭某个功能，或者对其进行修改，就必须修改所有相关方法。这不但增加了开发人员的工作量，而且提高了代码的出错率。

为了解决这一问题，AOP 思想随之产生。AOP 采取横向抽取机制，将分散在各个方法中的重复代码提取出来，然后在程序编译或运行时再将这些提取出来的代码应用到需要执行的地方。这种

采用横向抽取机制的方式，采用传统的 OOP 思想显然是无法办到的，因为 OOP 只能实现父子关系的纵向重用。虽然 AOP 是一种新的编程思想，但却不是 OOP 的替代品，它只是 OOP 的延伸和补充。

在 AOP 思想中，通过 Aspect（切面）可以分别在不同类的方法中加入事务、日志、权限和异常等功能。

AOP 的使用使开发人员在编写业务逻辑时可以专心于核心业务，而不用过多地关注于其他业务逻辑的实现，这不但提高了开发效率，而且增强了代码的可维护性。

目前流行的 AOP 框架有两个，分别为 Spring AOP 和 AspectJ。Spring AOP 使用纯 Java 实现，不需要专门的编译过程和类加载器，在运行期间通过代理方式向目标类植入增强的代码。AspectJ 是一个基于 Java 语言的 AOP 框架，从 Spring 2.0 开始，Spring AOP 引入了对 AspectJ 的支持，AspectJ 扩展了 Java 语言，提供了一个专门的编译器，在编译时提供横向代码的植入。

3.1.2　AOP 术语

在学习使用 AOP 之前，首先要了解一下 AOP 的专业术语。这些术语包括 Aspect、Joinpoint、Pointcut、Advice、Target Object、Proxy 和 Weaving，对于这些专业术语的解释，具体如下。

- Aspect（切面）：在实际应用中，切面通常是指封装的用于横向插入系统功能（如事务、日志等）的类，该类要被 Spring 容器识别为切面，需要在配置文件中通过<bean>元素指定。
- Joinpoint（连接点）：在程序执行过程中的某个阶段点，它实际上是对象的一个操作，例如方法的调用或异常的抛出。在 Spring AOP 中，连接点就是指方法的调用。
- Pointcut（切入点）：是指切面与程序流程的交叉点，即那些需要处理的连接点。通常在程序中，切入点指的是类或者方法名，如某个通知要应用到所有以 add 开头的方法中，那么所有满足这一规则的方法都是切入点。
- Advice（通知增强处理）：AOP 框架在特定的切入点执行增强处理，即在定义好的切入点处所要执行的程序代码。可以将其理解为切面类中的方法，它是切面的具体实现。
- Target Object（目标对象）：是指所有被通知的对象，也称为被增强对象。如果 AOP 框架采用的是动态的 AOP 实现，那么该对象就是一个被代理对象。
- Proxy（代理）：将通知应用到目标对象之后，被动态创建的对象。
- Weaving（织入）：将切面代码插入目标对象上，从而生成代理对象的过程。

3.2　AspectJ 开发

AspectJ 是一个基于 Java 语言的 AOP 框架，它提供了强大的 AOP 功能。Spring 2.0 以后，Spring AOP 引入了对 AspectJ 的支持，并允许直接使用 AspectJ 进行编程，而 Spring 自身的 AOP API 也尽量与 AspectJ 保持一致。新版本的 Spring 框架建议使用 AspectJ 来开发 AOP。

使用 AspectJ 实现 AOP 有两种方式：一种是基于 XML 的声明式 AspectJ；另一种是基于注解的

声明式 AspectJ。本节将对这两种 AspectJ 的开发方式进行讲解。

3.2.1 基于 XML 的声明式 AspectJ

基于 XML 的声明式 AspectJ 是指通过 XML 文件来定义切面、切入点及通知，所有的切面、切入点和通知都必须定义在<aop:config>元素内。Spring 配置文件中的<beans>元素下可以包含多个<aop:config>元素，一个<aop:config>元素中又可以包含属性和子元素，其子元素包括<aop:pointcut>、<aop:advisor>和<aop:aspect>。在配置时，这 3 个子元素必须按照此顺序来定义。在<aop:aspect>元素下，同样包含属性和多个子元素，通过使用<aop:aspect>元素及其子元素就可以在 XML 文件中配置切面、切入点和通知。常用元素的配置代码如下所示。

```
<!-- 定义切面 Bean -->
<bean id="myAspect" class="com.smm. aspectj.xmI.MyAspect />
<aop:config>
        <!-- 1.配置切面 -->
        <aop:aspect id="aspect" ref="myAspect">
            <!-- 2.配置切入点 -->
            <aop:pointcut expression="execution(* com.ssm.aspectj.*.*(..))" id="myPointCut"/>
            <!-- 3.配置通知 -->
                <!-- 前置通知 -->
                <aop:before method="myBefore" pointcut-ref="myPointCut" />
                <!--后置通知-->
                <aop:after-returning method="myAfterReturning"
                     pointcut-ref="myPointCut" returning="returnVal" />
                <!--环绕通知 -->
                <aop:around method="myAround" pointcut-ref="myPointCut" />
                <!--异常通知 -->
                <aop:after-throwing method="myAfterThrowing"
                      pointcut-ref="myPointCut" throwing="e" />
                <!--最终通知 -->
                <aop:after method="myAfter" pointcut-ref="myPointCut" />
        </aop:aspect>
    </aop:config>
```

为了让读者能够清楚地掌握上述代码中的配置信息，下面对上述代码的配置内容进行详细讲解。

1. 配置切面

在 Spring 的配置文件中，配置切面使用的是<aop:aspect>元素，该元素会将一个已定义好的 Spring Bean 转换成切面 Bean，所以要在配置文件中先定义一个普通的 Spring Bean（如上述代码中定义的 myAspect）。定义完成后，通过<aop:aspect>元素的 ref 属性即可引用该 Bean。

配置<aop:aspect>元素时，通常会指定 id 和 ref 两个属性，如表 3.1 所示。

表 3.1 <aop:aspect>元素的属性及其描述

属性名称	描述
id	用于定义该切面的唯一标识名称
ref	用于引用普通的 Spring Bean

2. 配置切入点

在 Spring 的配置文件中，切入点是通过<aop:pointcut>元素来定义的。当<aop:pointcut>元素作为<aop:config>元素的子元素定义时，表示该切入点是全局切入点，可以被多个切面所共享；当<aop:pointcut>元素作为<aop:aspect>元素的子元素时，表示该切入点只对当前切面有效。在定义<aop:pointcut>元素时，通常会指定 id 和 expression 两个属性，如表 3.2 所示。

表 3.2 <aop:pointcut>元素的属性及其描述

属性名称	描述
id	用于定义切入点的唯一标识名称
expression	用于指定切入点关联的切入点表达式

在上述配置代码片段中，execution(* com.ssm.jdk.*.*(..))就是定义的切入点表达式，该切入点表达式的意思是匹配 com.ssm.jdk 包中任意类的任意方法的执行。其中 execution 是表达式的主体，第 1 个*表示的是返回类型，使用*代表所有类型；com.ssm.jdk 表示的是需要拦截的包名，后面第 2 个*表示的是类名，使用*代表所有的类；第 3 个*表示的是方法名，使用*表示所有方法；后面的()表示方法的参数，其中的“..”表示任意参数。需要注意的是，第 1 个*与包名之间有一个空格。

上面示例中定义的切入点表达式只是开发中常用的配置方式，而 Spring AOP 中切入点表达式的基本格式如下:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?
```

在上述格式中，各部分说明如下：

- modifiers-pattern：表示定义的目标方法的访问修饰符，如 public、private 等。
- ret-type-pattern：表示定义的目标方法的返回值类型，如 void、String 等。
- declaring-type-pattern：表示定义的目标方法的类路径，如 com.ssm.jdk.UserDaoImpl。
- name-pattern：表示具体需要被代理的目标方法，如 add()方法。
- param-pattern：表示需要被代理的目标方法包含的参数，本章示例中目标方法参数都为空。
- throws- pattern：表示需要被代理的目标方法抛出的异常类型。

提 示
带有问号（?）的部分（如 modifiers-pattern、declaring-type-pattern 和 throws-pattern）表示可选配置项，其他部分属于必须配置项。

想要了解更多切入点表达式的配置信息，读者可以参考 Spring 官方文档的切入点声明部分（Declaring a pointcut）。

3. 配置通知

在配置代码中，分别使用<aop:aspect>的子元素配置了 5 种常用通知，这些子元素不支持再使用子元素，但在使用时可以指定一些属性，如表 3.3 所示。

表 3.3 通知的常用属性及其描述

属性名称	描述
pointcut	用于指定一个切入点表达式，Spring 将在匹配该表达式的连接点时植入该通知
pointcut-ref	指定一个已经存在的切入点名称，如配置代码中的 myPointcut。通常 pointcut 和 pointcut--ref 两个属性只需要使用其中之一
method	指定一个方法名，指定将切面 Bean 中的该方法转换为增强处理
throwing	只对<after-throwing>元素有效，用于指定一个形参名，异常通知方法可以通过该形参访问目标方法所抛出的异常
returning	只对<after-returning>元素有效，用于指定一个形参名，后置通知方法可以通过该形参访问目标方法的返回值

【示例 3-1】了解了如何在 XML 中配置切面、切入点和通知后，接下来通过一个案例来演示如何在 Spring 中使用基于 XML 的声明式 AspectJ，具体实现步骤如下。

（1）创建一个名为 chapter03 的动态 Web 项目，导入 Spring 构架所需求的 JAR 包到项目的 lib 目录中，并发布到类路径下。同时，导入 AspectJ 框架相关的 JAR 包，具体如下。

- spring- aspects-4.3.6.RELEASE.jar：Spring 为 AspectJ 提供的实现，Spring 的包中已经提供。
- aspectjweaver-1.8.10.jar：是 AspectJ 框架所提供的规范，读者可以通过网址"http://mvnrepository.com/artifact/org.aspectj/aspectjweaver/1.8.10"下载。

（2）在 chapter03 项目的 src 目录下创建一个 com.ssm.aspectj 包，在该包中创建接口 UserDao，并在接口中编写添加和删除的方法，如文件 3.1 所示。

文件 3.1 UserDao.java

```
package com.ssm.aspectj;
public interface UserDao {
  //添加用户方法
     public void addUser();
  //删除用户方法
  public void deleteUser();
}
```

（3）在 com.ssm.aspectj 包中创建 UserDao 接口的实现类 UserDaoImpl，该类需要实现接口中的方法，如文件 3.2 所示。

文件 3.2 UserDaoImpl.java

```
package com.ssm.aspectj;
public class UserDaoImpl implements UserDao {
    public void addUser() {
     System. out. println("添加用户");
    }
```

```
    public void deleteUser() {
     System.out.println("删除用户");
    }
}
```

本案例中将实现类 UserDaoImpl 作为目标类，对其中的方法进行增强处理。

（4）在 chapter03 项目的 src 目录下创建一个 com.ssm.aspectj.xml 包，在该包中创建切面类 MyAspect，并在类中分别定义不同类型的通知，如文件 3.3 所示。

文件 3.3 MyAspect.java

```
package com.ssm.aspectj.xml;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
/**
 * 切面类，在此类中编写通知
 */
public class MyAspect {
    //前置通知
    public void myBefore(JoinPoint joinPoint){
     System.out.print("前置通知：模拟执行权限检查...，");
     System.out.print("目标类是："+joinPoint.getTarget());
     System.out.println("，被植入增强处理的目标方法为："+
                      joinPoint.getSignature().getName());
    }
    //后置通知
    public void myAfterReturning(JoinPoint joinPoint) {
     System.out.print("后置通知：模拟记录日志...，");
     System.out.println("被植入增强处理的目标方法为：" +
                       joinPoint.getSignature().getName());
    }
    /**
     * 环绕通知
     * ProceedingJoinPoint 是 JoinPoint 的子接口，表示可执行目标方法
     * 1.必须是 Object 类型的返回值
     * 2.必须接收一个参数，类型为 ProceedingJoinPoint
     * 3.必须是 throws Throwable
     */
    public Object myAround(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
     //开始
     System.out.println("环绕开始：执行目标方法之前，模拟开启事务...，");
     //执行当前目标方法
     Object obj=proceedingJoinPoint.proceed();
     //结束
     System.out.println("环绕结束：执行目标方法之后，模拟关闭事务...，");
     return obj;
    }
```

```
    //异常通知
    public void myAfterThrowing(JoinPoint joinPoint,Throwable e){
     System.out.println("异常通知：出错了"+e.getMessage());
    }
    //最终通知
    public void myAfter(){
     System.out.println("最终通知：模拟方法结束后释放资源...");
    }
}
```

在文件 3.3 中，分别定义了 5 种不同类型的通知，在通知中使用了 JoinPoint 接口及其子接口 ProceedingJoinPoint 作为参数来获得目标对象的类名、目标方法名和目标方法参数等。

注　意

环绕通知必须接收一个类型为 ProceedingJoinPoint 的参数，返回值也必须是 Object 类型，且必须抛出异常。异常通知中可以传入 Throwable 类型的参数来输出异常信息。

（5）在 com.ssm.aspectj.xml 包中创建配置文件 applicationContext.xml，并编写相关配置，如文件 3.4 所示。

文件 3.4　applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!-- 1 目标类 -->
    <bean id="userDao" class="com.ssm.aspectj.UserDaoImpl" />
    <!-- 2 切面 -->
    <bean id="myAspect" class="com.ssm.aspectj.xml.MyAspect" />
    <!-- 3 aop 编程 -->
    <aop:config>
     <!-- 1.配置切面 -->
     <aop:aspect id="aspect" ref="myAspect">
         <!-- 2.配置切入点 -->
         <aop:pointcut expression="execution(* com.ssm.aspectj.*.*(..))" id="myPointCut" />
         <!-- 3.配置通知 -->
              <!-- 前置通知 -->
              <aop:before method="myBefore" pointcut-ref="myPointCut" />
              <!--后置通知-->
              <aop:after-returning method="myAfterReturning" pointcut-ref="myPointCut"
                  returning="returnVal"/>
```

```
            <!--环绕通知 -->
            <aop:around method="myAround" pointcut-ref="myPointCut" />
            <!--异常通知 -->
            <aop:after-throwing method="myAfterThrowing" pointcut-ref="myPointCut"
                throwing="e" />
            <!--最终通知 -->
            <aop:after method="myAfter" pointcut-ref="myPointCut" />
        </aop:aspect>
       </aop:config>
</beans>
```

注　意

在 AOP 的配置信息中，使用<aop:after-returning>配置的后置通知和使用<aop:after>配置的最终通知虽然都是在目标方法执行之后执行，但它们是有区别的。后置通知只有在目标方法成功执行后才会被植入，而最终通知不论目标方法如何结束（包括成功执行和异常中止两种情况），它都会被植入。另外，如果程序没有异常，异常通知将不会执行。

（6）在 com.ssm.aspectj.xml 包下创建测试类 TestXmlAspectJ，在类中为了更加清晰地演示几种通知的执行情况，这里只对 addUser()方法进行增强测试，如文件 3.5 所示。

文件 3.5　TestXmlAspectJ.java

```java
package com.ssm.aspectj.xml;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.ssm.aspectj.UserDao;
public class TestXmlAspectJ {
    public static void main(String[] args) {
      // 定义配置文件路径
     String xmlPath="com/ssm/aspectj/xml/applicationContext.xml";
      // 初始化 Spring 容器，加载配置文件
     ApplicationContext applicationContext=new ClassPathXmlApplicationContext(xmlPath);
     // 从容器中获得 userDao 实例
     UserDao userDao=(UserDao)applicationContext.getBean("userDao");
     // 执行添加用户方法
     userDao.addUser();
    }
}
```

执行程序后，控制台的输出结果如图 3.1 所示。

```
Markers  Properties  Servers  Snippets  Problems  Console  JUnit  Terminal
<terminated> TestXmlAspectJ [Java Application] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年10月20日 下午11:39:45)
前置通知：模拟执行权限检查...，目标类是：com.ssm.aspectj.UserDaoImpl@5de723，被植入增强处理的目标方法为：addUser
环绕开始：执行目标方法之前，模拟开启事务...，添加用户
最终通知：模拟方法结束后释放资源..环绕结束：执行目标方法之后，模拟关闭事务...，后置通知：模拟记录日志...，被植入增强处理的目标方法为：addUser
```

图 3.1　运行结果 1

要查看异常通知的执行效果，可以在 UserDaoImpl 类的 addUser()方法中添加出错代码，如“int i=10/0;”。重新运行测试类，将可以看到异常通知的执行，此时控制台的输出结果如图 3.2 所示。



图 3.2　运行结果 2

从图 3.1 和图 3.2 可以看出，使用基于 XML 的声明式 AspectJ 已经实现了 AOP 开发。

3.2.2　基于注解的声明式 AspectJ

基于 XML 的声明式 AspectJ 实现 AOP 编程虽然便捷，但是存在一些缺点，那就是要在 Spring 文件中配置大量的代码信息。为了解决这个问题，AspectJ 框架为 AOP 的实现提供了一套注解，用以取代 Spring 配置文件中为实现 AOP 功能所配置的臃肿代码。

关于 AspectJ 注解的介绍如表 3.4 所示。

表 3.4　AspectJ 的注解及其描述

注解名称	描述
@Aspect	用于定义一个切面
@Pointcut	用于定义切入点表达式。在使用时还需定义一个包含名字和任意参数的方法签名来表示切入点名称。实际上，这个方法签名就是一个返回值为 void 且方法体为空的普通方法
@Before	用于定义前置通知，相当于 BeforeAdvice。在使用时，通常需要指定一个 value 属性值，该属性值用于指定一个切入点表达式（可以是已有的切入点，也可以直接定义切入点表达式）
@AfterReturning	用于定义后置通知，相当于 AfterReturningAdvice。在使用时可以指定 pointcut/value 和 returning 属性，其中 pointcut/value 两个属性的作用一样，都用于指定切入点表达式，returning 属性值用于表示 Advice()方法中可定义与此同名的形参，该形参可用于访问目标方法的返回值
@Around	用于定义环绕通知，相当于 MethodInterceptor。在使用时需要指定一个 value 属性，该属性用于指定通知被植入的切入点
@AfterThrowing	用于定义异常通知来处理程序中未处理的异常，相当于 ThrowAdvice。在使用时可指定 pointcut/value 和 throwing 属性。其中，pointcut/value 用于指定切入点表达式，而 throwing 属性值用于指定一个形参名来表示 Advice()方法中可定义与此同名的形参，该形参可用于访问目标方法抛出的异常
@After	用于定义最终 final 通知，无论是否有异常，该通知都会执行。使用时需要指定一个 value 属性，用于指定该通知被植入的切入点
@DeclareParents	用于定义引介通知，相当于 IntroductionInterceptor（不要求掌握）

【示例 3-2】为了使读者可以快速地掌握这些注解，接下来重新使用注解的形式实现 3.2.1 小节的案例，具体步骤如下。

（1）在 chapter03 项目的 src 目录下创建 com.ssm.aspectj.annotation 包，将文件 3.3 的切面类 MyAspect 复制到该包下，并对该文件进行修改，如文件 3.6 所示。

文件 3.6　MyAspect.java

```
package com.ssm.aspectj.annotation;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;
/**
 * 切面类，在此类中编写通知
 */
@Aspect
@Component
public class MyAspect {
    //定义切入点表达式
    @Pointcut("execution(* com.ssm.aspectj.*.*(..))")
    //使用一个返回值为 void、方法体为空的方法来命名切入点
    public void myPointCut(){}
    //前置通知
    @Before("myPointCut()")
    public void myBefore(JoinPoint joinPoint){
     System.out.print("前置通知：模拟执行权限检查..，");
     System.out.print("目标类是："+joinPoint.getTarget());
     System.out.println("，被植入增强处理的目标方法为："+
            joinPoint.getSignature().getName());
    }
    //后置通知
    @AfterReturning(value="myPointCut()")
    public void myAfterReturning(JoinPoint joinPoint) {
     System.out.print("后置通知：模拟记录日志..，");
     System.out.println("被植入增强处理的目标方法为：" +
            joinPoint.getSignature().getName());
    }
    /**
     * 环绕通知
     * ProceedingJoinPoint 是 JoinPoint 的子接口，表示可执行目标方法
     * 1.必须是 Object 类型的返回值
     * 2.必须接收一个参数，类型为 ProceedingJoinPoint
     * 3.必须 throws Throwable
```

```
     */
    @Around("myPointCut()")
    public Object myAround(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
     //开始
     System.out.println("环绕开始：执行目标方法之前，模拟开启事务..，");
     //执行当前目标方法
     Object obj=proceedingJoinPoint.proceed();
     //结束
     System.out.println("环绕结束：执行目标方法之后，模拟关闭事务..，");
     return obj;
    }
    //异常通知
    @AfterThrowing(value="myPointCut()",throwing="e")
    public void myAfterThrowing(JoinPoint joinPoint,Throwable e){
     System.out.println("异常通知：出错了"+e.getMessage());
    }
    //最终通知
    @After("myPointCut()")
    public void myAfter(){
     System.out.println("最终通知：模拟方法结束后释放资源..");
    }
}
```

在文件 3.6 中，首先使用@Aspect 注解定义了切面类，由于该类在 Spring 中是作为组件使用的，因此还需要添加@Component 注解才能生效。然后使用@Pointcut 注解来配置切入表达式，并通过定义方法来表示切入点名称。接下来在每个通知相应的方法上添加了相应的注解，并将切入点名称“myPointcut”作为参数传递给需要执行增强的通知方法。如果需要其他参数（如异常通知的异常参数），可以根据代码提示传递相应的属性值。

（2）在目标类 com.ssm.aspectj.UserDaoImpl 中添加注解@Repository("userDao")。

（3）在 com.ssm.aspectj.annotation 包下创建配置文件 applicationContext.xml，并对该文件进行编辑，如文件 3.7 所示。

文件 3.7　applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 指定需要扫描的包，使注解生效 -->
```

```
    <context:component-scan base-package="com.ssm" />
    <!-- 启动基于注解的声明式 AspectJ 支持 -->
    <aop:aspectj-autoproxy />
</beans>
```

在文件 3.7 中，首先引入了 context 约束信息，然后使用<context>元素设置了需要扫描的包，使注解生效。由于此案例中的目标类位于 com.ssm.aspectj 包中，因此这里设置 base-package 的值为 “com.ssm"。最后，使用<aop. aspectj-autoproxy /> 来启动 Spring 对基于注解的声明式 Aspect 的支持。

（4）在 com.ssm.aspectj.annotation 包中创建测试类 TestAnnotation，该类与文件 3.5 基本一致，只是配置文件的路径有所不同，如文件 3.8 所示。

文件 3.8 TestAnnotation.java

```
package com.ssm.aspectj.annotation;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.ssm.aspectj.UserDao;
public class TestAnnotation {
    public static void main(String[] args) {
     String xmlPath="com/ssm/aspectj/annotation/applicationContext.xml";
     ApplicationContext applicationContext=new ClassPathXmlApplicationContext(xmlPath);
     //从容器中获得 userDao 实例
     UserDao userDao=(UserDao)applicationContext.getBean("userDao");
     //执行添加用户方法
     userDao.addUser();
    }
}
```

执行程序后，控制台的输出结果如图 3.3 所示。

```
Markers  Properties  Servers  Snippets  Problems  Console  JU JUnit  Terminal
<terminated> TestAnnotation [Java Application] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年10月21日 下午1:30:08)
环绕开始：执行目标方法之前，模拟开启事务...
前置通知：模拟执行权限检查...，目标类是：com.ssm.aspectj.UserDaoImpl@dddf46，被植入增强处理的目标方法为：addUser
添加用户
环绕结束：执行目标方法之后，模拟关闭事务...
最终通知：模拟方法结束后释放资源..
后置通知：模拟记录日志...，被植入增强处理的目标方法为：addUser
```

图 3.3 运行结果

在 UserDaoImpl 类的 addUser()方法中加上出错代码来演示异常通知的执行，控制台的输出结果如图 3.3 所示。

从图 3.2 和图 3.3 可以看出，基于注解的方式与基于 XML 的方式执行结果相同，只是在目标方法前后通知的执行顺序发生了变化。相对来说，使用注解的方式更加简单、方便，所以在实际开发中推荐使用注解的方式进行 AOP 开发。

注　意

如果在同一个连接点有多个通知需要执行，那么在同一切面中，目标方法之前的前置通知和环绕通知的执行顺序是未知的，目标方法之后的后置通知和环绕通知的执行顺序也是未知的。

3.3 习　　题

1．请列举你所知道的 AOP 专业术语并解释。
2．请列举你所知道的 Spring 的通知类型并解释。

第 4 章

Spring 的数据库开发

Spring 框架降低了 JavaEE API 的难度，其中包括 JDBC 的使用难度。JDBC 是 Spring 数据访问和集成中的重要模块，本章将详细讲解 Spring 中的 JDBC 知识。

本章主要涉及的知识点如下：

- Spring JDBC：Spring JdbcTemplate 的解析和 Spring JDBC 的配置。
- Spring JdbcTemplate 的常用方法：execute()、update()和 query()方法。

4.1 Spring JDBC

Spring 的 JDBC 模块负责数据库资源管理和错误处理，大大简化了开发人员对数据库的操作，使得开发人员可以从烦琐的数据库操作中解脱出来，从而将更多的精力投入编写业务逻辑中。

4.1.1 Spring JdbcTemplate 的解析

针对数据库的操作，Spring 框架提供了 JdbcTemplate 类，该类是 Spring 框架数据抽象层的基础，其他更高层次的抽象类是构建于 JdbcTemplate 类之上的。可以说，JdbcTemplate 类是 Spring JDBC 的核心类。

JdbcTemplate 类的继承关系十分简单。它继承自抽象类 JdbcAccessor，同时实现了 JdbcOperations 接口。

（1）JdbcOperations 接口定义了在 JdbcTemplate 类中可以使用的操作集合，包括添加、修改、查询和删除等操作。

（2）JdbcTemplate 类的直接父类是 JdbcAccessor，该类为子类提供了一些访问数据库时使用的

公共属性，具体如下。

- DataSource：其主要功能是获取数据库连接，具体实现时还可以引入对数据库连接的缓冲池和分布事务的支持，它可以作为访问数据库资源的标准接口。
- SQLExceptionTranslator：org.springframework.jdbc.support.SQLExceptionTranslator 接口负责对 SQLException 进行转译工作。通过必要的设置或者获取 SQLExceptionTranslator 中的方法可以使 JdbcTemplate 在需要处理 SQLException 时委托 SQLExceptionTranslator 的实现类来完成相关的转译工作。

4.1.2 Spring JDBC 的配置

Spring JDBC 模块主要由 4 个包组成，分别是 core（核心包）、dataSource（数据包）、object（对象包）和 support（支持包）。关于这 4 个包的具体说明如表 4.1 所示。

表 4.1 Spring JDBC 中的主要包及说明

包名	说明
core	包含 JDBC 的核心功能，包括 JdbcTemplate 类、SimpleJdbcInsert 类、SimpleJdbcCall 类以及 NamedParameterJdbcTemplate 类
dataSource	访问数据源的实用工具类，它有多种数据源的实现，可以在 Java EE 容器外部测试 JDBC 代码
object	以面向对象的方式访问数据库，它允许执行查询并将返回结果作为业务对象，可以在数据表的列和业务对象的属性之间映射查询结果
support	包含 core 和 object 包的支持类，例如提供异常转换功能的 SQLException 类

从表 4.1 可以看出，Spring 对数据库的操作都封装在了这几个包中，如果想要使用 Spring JDBC，就需要对其进行配置。在 Spring 中，JDBC 的配置是在配置文件 applicationContext.xml 中完成的，其配置模板如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!--1 配置数据源 -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--数据库驱动 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!--连接数据库的 url -->
        <property name="url" value="jdbc:mysql://localhost:3306/db_spring" />
        <!--连接数据库的用户名 -->
        <property name="username" value="root" />
        <!--连接数据库的密码 -->
        <property name="password" value="root" />
```

```
    </bean>
    <!--2 配置 JDBC 模板 -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--默认必须使用数据源 -->
        <property name="dataSource" ref="dataSource" />
    </bean>
    <!--3 配置注入类 -->
    <bean id="xxx" class="Xxx">
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
</beans>
```

在上述代码中定义了 3 个 Bean，分别是 dataSource、jdbcTemplate 和需要注入类的 Bean。其中 dataSource 对应的 org.springframework.jdbc.datasource.DriverManagerDataSource 类用于对数据源进行配置，jdbcTemplate 对应的 org.springframework.jdbc.core.JdbcTemplate 类中定义了 JdbcTemplate 的相关配置。上述代码中 dataSource 的配置就是 JDBC 连接数据库时所需的 4 个属性，如表 4.2 所示。

表 4.2　dataSource 的 4 个属性

属性名	含义
driverClassName	所使用的驱动名称，对应驱动 JAR 包中的 Driver 类
url	数据源所在地址
username	访问数据库的用户名
password	访问数据库的密码

表 4.2 中的 4 个属性需要根据数据库类型或者机器配置的不同设置相应的属性值。例如，如果数据库类型不同，就需要更改驱动名称；如果数据库不在本地，就需要将地址中的 localhost 替换成相应的主机 IP；如果修改过 MySQL 数据库的端口号（默认为 3306），就需要加上修改后的端口号，如果未修改，那么端口号可以省略；同时连接数据库的用户名和密码需要与数据库创建时设置的用户名和密码保持一致。本示例中 Spring 数据库的用户名和密码都是 root。

定义 jdbcTemplate 时，需要将 dataSource 注入 jdbcTemplate 中，而其他需要使用 jdbcTemplate 的 Bean，也需要将 jdbcTemplate 注入该 Bean 中（通常注入 Dao 类中，在 Dao 类中进行与数据库的相关操作）。

4.2　Spring JdbcTemplate 的常用方法

在 JdbcTemplate 类中提供了大量更新和查询数据库的方法，我们就是使用这些方法来操作数据库的。本节将介绍这些方法的使用。

4.2.1 execute()——执行 SQL 语句

execute(String sql)方法能够完成执行 SQL 语句的功能。

【示例 4-1】下面以创建数据表的 SQL 语句为例演示此方法的使用，具体步骤如下。

步骤01 在 MySQL 中创建一个名为 db_spring 的数据库，如图 4.1 所示。

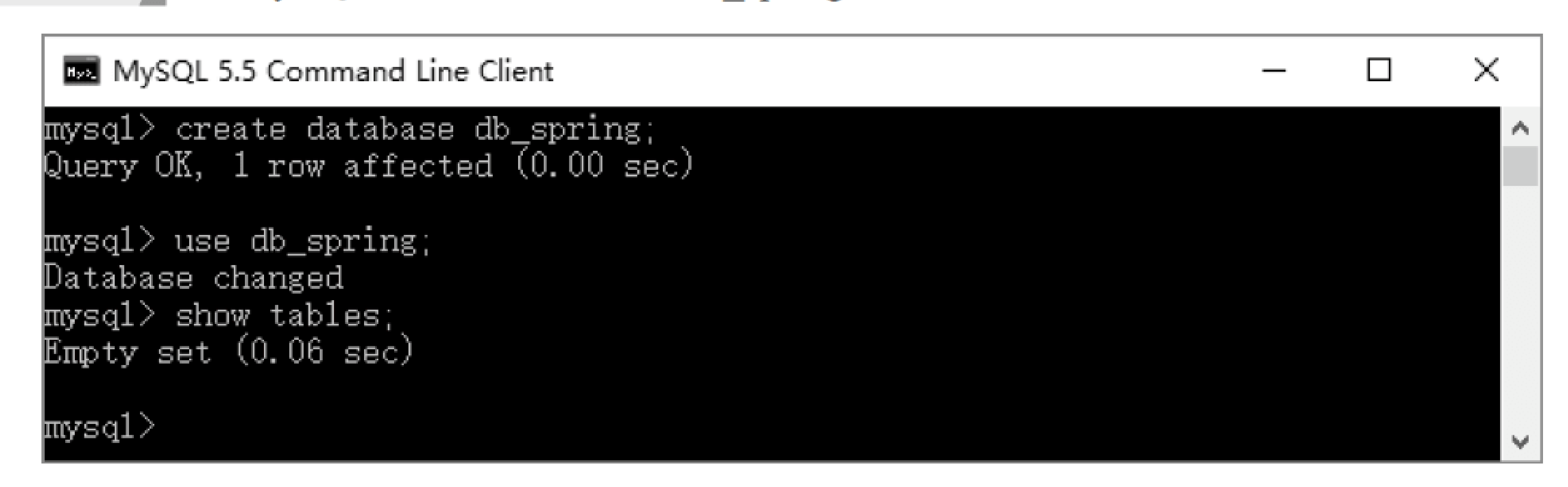


图 4.1 创建数据库

在图 4.1 中，首先使用 SQL 语句创建了数据库 db_spring，然后选择使用 db_spring。为了便于后续验证数据表是通过 execute(String sql)方法执行创建的，这里使用了 show tables 语句查看数据库中的表，其结果显示为空。

步骤02 在 Eclipse 中创建一个名为 chapter04 的 Web 项目，将运行 Spring 框架所需的 5 个基础 JAR 包、MySOL 数据库的驱动 JAR 包、Spring JDBC 的 JAR 包以及 Spring 事务处理的 JAR 包复制到项目的 lib 目录，并发布到类路径中。项目中所添加的 JAR 包如图 4.2 所示。

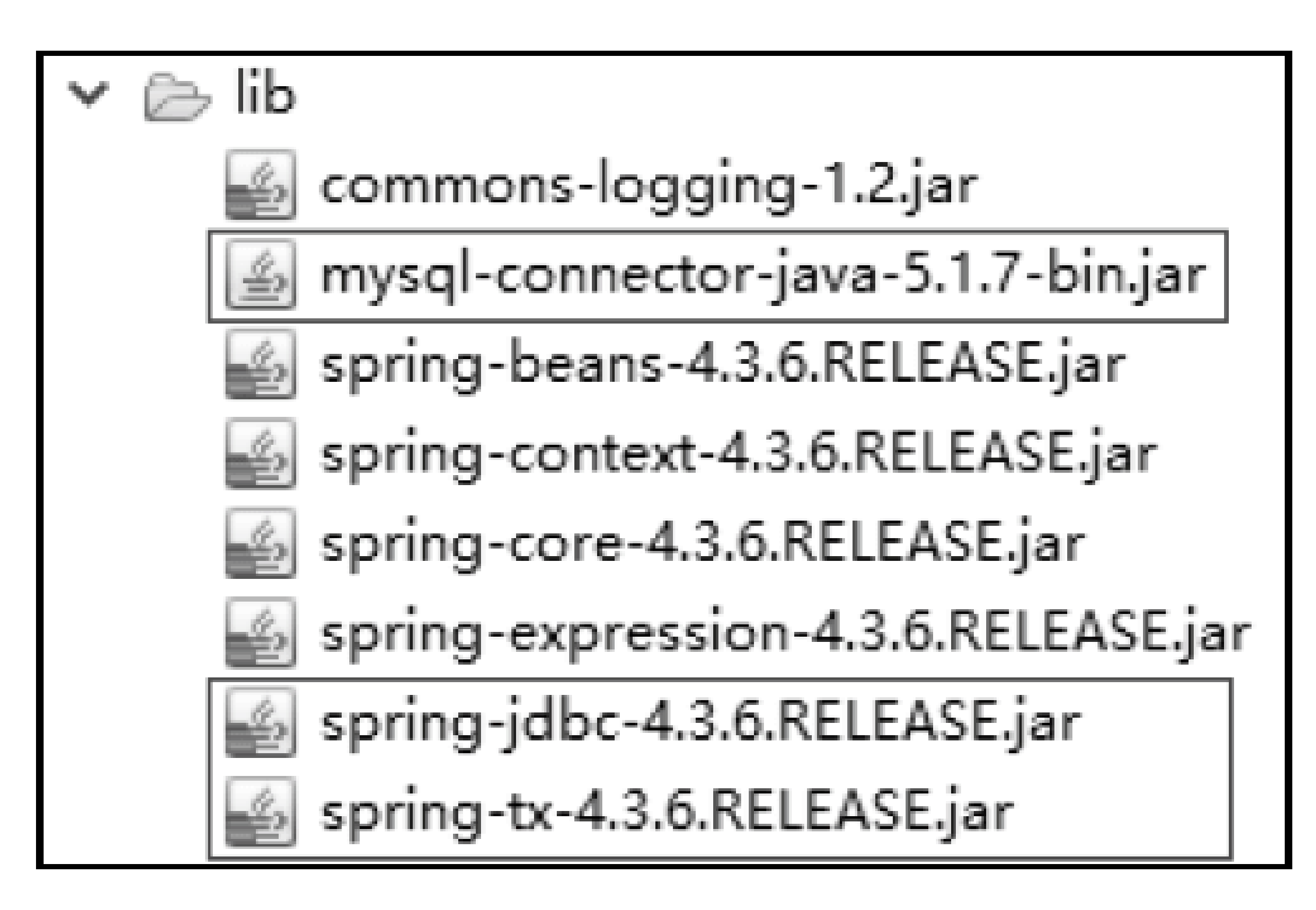


图 4.2 Spring JDBC 操作相关的 JAR 包

步骤03 在 src 目录下创建配置文件 applicationContext.xml，在该文件中配置 id 为 dataSource 的数据源 Bean 和 id 为 jdbcTemplate 的 JDBC 模板 Bean，并将数据源注入 JDBC 模板中，如文件 4.1 所示。

文件 4.1 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!--1 配置数据源 -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--数据库驱动 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!--连接数据库的 url -->
        <property name="url" value="jdbc:mysql://localhost:3306/db_spring" />
        <!--连接数据库的用户名 -->
        <property name="username" value="root" />
        <!--连接数据库的密码 -->
        <property name="password" value="root" />
    </bean>
    <!--2 配置 JDBC 模板 -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--默认必须使用数据源 -->
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

步骤04 在 src 目录下创建一个 com.ssm.jdbc 包，在该包中创建测试类 JdbcTemplateTest。在该类的 main()方法中通过 Spring 容器获取在配置文件中定义的 JdbcTemplate 实例，然后使用实例的 execute(String s)方法执行创建数据表的 SQL 语句，如文件 4.2 所示。

文件 4.2　JdbcTemplateTest.java

```
package com.ssm.jdbc;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
/**
 *使用 excute()方法创建表
 */
public class JdbcTemplateTest {
 public static void main(String[] args) {
     //加载配置文件
     ApplicationContext applicationContext =
                   new ClassPathXmlApplicationContext("applicationContext.xml");
     //获取 JdbcTemplate 实例
     JdbcTemplate jdbcTemplate =
                   (JdbcTemplate) applicationContext.getBean("jdbcTemplate");
     //使用 execute()方法执行 SQL 语句，创建用户表 user
     jdbcTemplate.execute("create table user(" +
                     "id int primary key auto_increment," +
                       "username varchar(40)," +
                     "password varchar(40))");
 }
```

```
    }
```

成功运行程序后，再次查询 db_spring 数据库，其结果如图 4.3 所示。从中可以看出，程序使用 execute(String sql)方法执行的 SQL 语句已成功创建了数据表 user。

```
MySQL 5.5 Command Line Client
mysql> show tables;
+-------------------+
| Tables_in_db_spring |
+-------------------+
| user              |
+-------------------+
1 row in set (0.00 sec)

mysql>
```

图 4.3　db_spring 数据库中的表

4.2.2　update()——更新数据

update()方法可以完成插入、更新和删除数据的操作。在 JdbcTemplate 类中提供了一系列 update()方法，其常用格式如表 4.3 所示。

表 4.3　JdbcTemplate 类中常用的 update()方法

方法	说明
int update(String sql)	该方法是最简单的 update 方法重载形式，直接执行传入的 SQL 语句，并返回受影响的行数
int update(PreparedStatementCreator psc)	该方法执行从 PreparedStatementCreator 返回的语句，然后返回受影响的行数
int update(String sql, PreparedStatementSetter pss)	该方法通过 PreparedStatementsetter 设置 SQL 语句中的参数，并返回受影响的行数
int update(String sql, Object... args)	该方法使用 Object...设置 SQL 语句中的参数，要求参数不能为 NULL，并返回受影响的行数

【示例 4-2】通过一个用户管理的案例来演示 update()方法的使用，具体步骤如下。

步骤 01 在 chapter04 项目的 com.ssm.jdbc 包中创建 User 类，在该类中定义 id、username 和 password 属性，以及其对应的 getter()/ setter()方法，如文件 4.3 所示。

文件 4.3　User.java

```
package com.ssm.jdbc;
// User 实例类
public class User {
 private Integer id;  // 用户 id
 private String username; // 用户名
 private String password; // 密码
 public Integer getId() {
     return id;
```

```
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String toString() {
        return "User [id=" + id + ", username=" + username + ", password=" + password + "]";
    }
}
```

步骤02 在 com.ssm.jdbc 包中创建接口 UserDao，并在接口中定义添加、更新和删除用户的方法，如文件 4.4 所示。

文件 4.4 UserDao.java

```
package com.ssm.jdbc;
public interface UserDao {
 // 添加用户方法
 public int addUser(User user);
 // 更新用户方法
 public int updateUser(User user);
 // 删除用户方法
 public int deleteUser(int id);
}
```

步骤03 在 com.ssm.jdbc 包中创建 UserDao 接口的实现类 UserDaoImpl，并在类中实现添加、更新和删除账户的方法，编辑后如文件 4.5 所示。

文件 4.5 UserDaoImpl.java

```
package com.ssm.jdbc;
import org.springframework.jdbc.core.JdbcTemplate;
public class UserDaoImpl implements UserDao {
 private JdbcTemplate jdbcTemplate;
 public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
     this.jdbcTemplate = jdbcTemplate;
 }
```

```
    // 添加用户方法
    public int addUser(User user) {
        String sql="insert into user(username,password) value(?,?)";
        Object[] obj=new Object[]{
                user.getUsername(),
                user.getPassword()
        };
        int num=this.jdbcTemplate.update(sql,obj);
        return num;
    }
    // 更新用户方法
    public int updateUser(User user) {
        String sql="update user set username=?,password=? where id=?";
        Object[] params=new Object[]{
                user.getUsername(),
                user.getPassword(),
                user.getId()
        };
        int num=this.jdbcTemplate.update(sql,params);
        return num;
    }
    // 删除用户方法
    public int deleteUser(int id) {
        String sql="delete from user where id=?";
        int num=this.jdbcTemplate.update(sql,id);
        return num;
    }
}
```

从上述 3 种操作的代码可以看出，添加、更新和删除操作的实现步骤类似，只是定义的 SQL 语句有所不同。

步骤04 在 applicationContext.xml 中定义一个 id 为 userDao 的 Bean，该 Bean 用于将 jdbcTemplate 注入 userDao 实例中，其代码如下所示。

```
<!-- 定义 id 为 userDao 的 Bean -->
    <bean id="userDao" class="com.ssm.jdbc.UserDaoImpl">
        <!--将 jdbcTemplate 注入 userDao 实例中 -->
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
```

步骤05 在测试类 JdbcTemplateTest 中添加一个测试方法 addUserTest()。该方法主要用于添加用户信息，其代码如下所示。

```
 // 测试添加用户方法
@Test
    public void addUserTest(){
    //加载配置文件
```

```
        ApplicationContext applicationContext=
                  new ClassPathXmlApplicationContext("applicationContext.xml");
    //获取 userDao 实例
        UserDao userDao=(UserDao)applicationContext.getBean("userDao");
    //创建 user 实例
        User user = new User();
    //设置 user 实例属性值
        user.setUsername("zhangsan");
        user.setPassword("123456");
    //添加用户
        int num=userDao.addUser(user);
        if(num>0){
            System.out.println("成功插入了"+num+"条数据。");
        }else{
            System.out.println("插入操作执行失败。");
        }
    }
```

在上述代码中，获取 UserDao 的实例后又创建了 User 对象，并向 User 对象中添加了属性值。然后调用 UserDao 对象的 addUser()方法向数据表中添加一条数据。最后，通过返回的受影响的行数来判断数据是否插入成功。

使用 JUnit4 测试运行后，控制台的输出结果如图 4.4 所示。

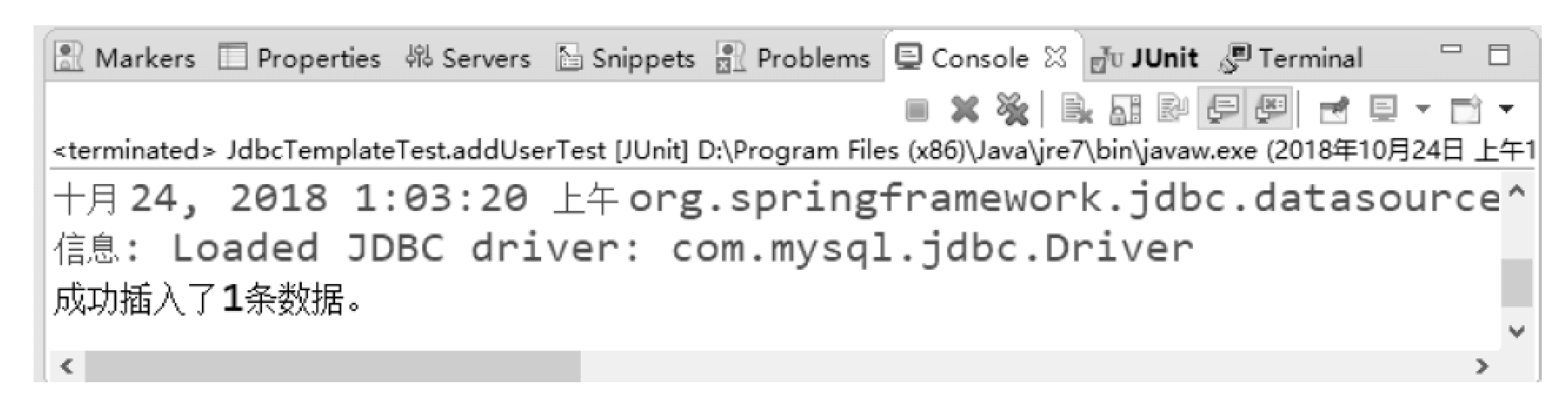


图 4.4　运行结果

此时再次查询数据库中的 user 表，其结果如图 4.5 所示。从中可以看出，使用 JdbcTemplate 的 update()方法已成功地向数据表中插入了一条数据。

```
MySQL 5.5 Command Line Client
mysql> use db_spring;
Database changed
mysql> select * from user;
+----+----------+----------+
| id | username | password |
+----+----------+----------+
|  1 | zhangsan | 123456   |
+----+----------+----------+
1 row in set (0.05 sec)

mysql> _
```

图 4.5　运行结果

步骤06 执行完插入操作后，接下来使用 JdbcTemplate 类的 update()方法执行更新操作。在测试类 JdbcTemplateTest 中添加一个测试方法 updateUser Test()，其代码如下所示。

```
@Test
    public void updateUserTest(){
        ApplicationContext applicationContext=
                new ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao=(UserDao)applicationContext.getBean("userDao");
        User user = new User();
        user.setId(1);
        user.setUsername("tom");
        user.setPassword("111111");
    //更新用户
        int num=userDao.updateUser(user);
        if(num>0){
            System.out.println("成功更新了"+num+"条数据。");
        }else{
            System.out.println("更新操作执行失败。");
        }
    }
```

与 addUserTest()方法相比，更新操作的代码增加了 id 属性值的设置，并在将用户名和密码修改后调用了 UserDao 对象中的 updateUser()方法执行对数据表的更新操作。使用 JUnit4 运行方法后，再次查询数据库中的 user 表，其结果如图 4.6 所示。从中可以看出，使用 update()方法已成功更新了 user 表中 id 为 1 的用户的用户名和密码。



图 4.6 运行结果

步骤07 在测试类 JdbcTemplateTest 中添加一个测试方法 deleteUserTest()来执行删除操作，其代码如下所示。

```
@Test
    public void deleteUserTest(){
        ApplicationContext applicationContext=
                new ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao=(UserDao)applicationContext.getBean("userDao");
    //删除用户
        int num=userDao.deleteUser(1);
        if(num>0){
```

```
            System.out.println("成功删除了"+num+"条数据。");
        }else{
            System.out.println("删除操作执行失败。");
        }
    }
```

在上述代码中，获取了 UserDao 的实例后，执行实例中的 deleteUser()方法来删除 id 为 1 的数据。

使用 JUnit4 测试运行方法后，查询 user 表中的数据，其结果如图 4.7 所示。从中可以看出，已成功通过 deleteUser()方法删除了 id 为 1 的数据。由于 user 表中只有一条数据，因此删除后表中数据为空。

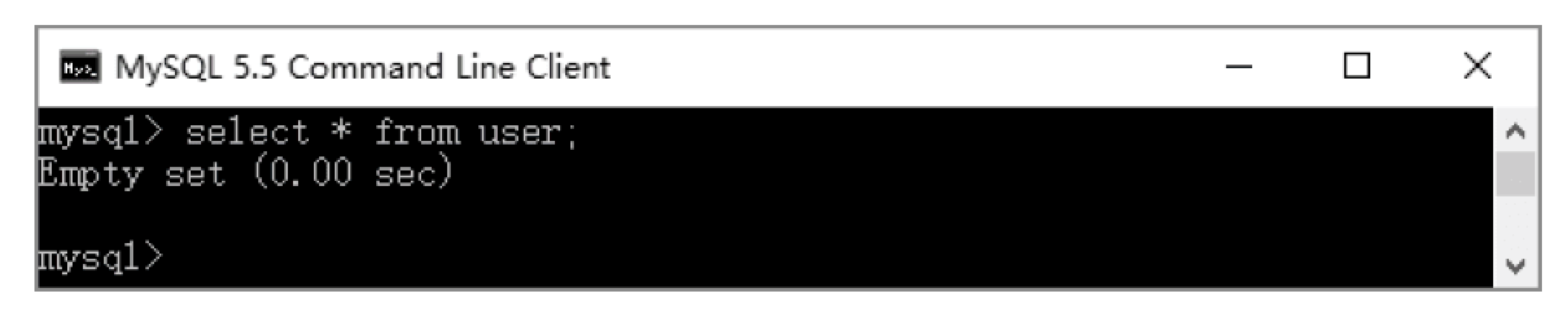


图 4.7　运行结果

4.2.3　query()——查询数据

JdbcTemplate 类中还提供了大量的 query()方法来处理各种对数据库表的查询操作。其中常用的几个 query()方法格式如表 4.4 所示。

表 4.4　JdbcTemplate 中常用的 query()方法

方法	说明
List query(String sql,RowMapper rowMapper)	执行 String 类型参数提供的 SQL 语句，并通过 RowMapper 返回一个 List 类型的结果
List query(String sql,PreparedStatementSetter pss,RowMapper rowMapper)	根据 String 类型参数提供的 SQL 语句创建 PreparedStatement 对象，通过 RowMapper 将结果返回到 List 中
List query(String sql,RowMapper rowMapper)	使用 Object 的值来设置 SQL 语句中的参数值，采用 RowMapper 回调方法可以直接返回 List 类型的数据
queryForObject(String sql, RowMapper rowMapper,Object...args)	将 args 参数绑定到 SQL 语句中，并通过 RowMapper 返回一个 Object 类型的单行记录
queryForList(string sql, Object[] args, class<T> elementType)	该方法可以返回多行数据的结果，但必须是返回列表，elementType 参数返回的是 List 元素类型

【示例 4-3】通过一个具体的案例演示 query()方法的使用，其实现步骤如下。

步骤01 向数据表 user 中插入几条数据，插入后 user 表中的数据如图 4.8 所示。



图 4.8 运行结果

步骤02 在 UserDao 中分别创建一个通过 id 查询单个用户和查询所有用户的方法，其代码如下所示。

```
    //通过 id 查询用户
    public User findUserById(int id);
    //查询所有用户
    public List<User> findAllUser();
```

步骤03 在 UserDao 接口的实现类 UserDaoImpl 中实现接口中的方法，并使用 query()方法分别进行查询，其代码如下所示。

```
    //通过 id 查询用户数据信息
    public User findUserById(int id) {
        String sql="select * from user where id=?";
        RowMapper<User> rowMapper=new BeanPropertyRowMapper<User>(User.class);
        return this.jdbcTemplate.queryForObject(sql,rowMapper,id);
    }
    //查询所有用户数据信息
    public List<User> findAllUser() {
        String sql="select * from user";
        RowMapper<User> rowMapper=new BeanPropertyRowMapper<User>(User.class);
        return this.jdbcTemplate.query(sql,rowMapper);
    }
```

在上面两个方法代码中，BeanPropertyRowMapper 是 RowMapper 接口的实现类，可以自动地将数据表中的数据映射到用户自定义的类中（前提是用户自定义类中的字段要与数据表中的字段相对应）。创建完 BeanPropertyRowMapper 对象后，在 findUserById()方法中通过 queryForObject()方法返回了一个 Object 类型的单行记录，而在 findAllUser()方法中通过 query()方法返回了一个结果集合。

步骤04 在测试类 JdbcTemplateTest 中添加一个测试方法 findUserByIdTest()来测试条件查询，其代码如下所示。

```
@Test
    public void findUserByIdTest(){
        ApplicationContext applicationContext=
```

```
                new ClassPathXmlApplicationContext("applicationContext.xml");
    //通过 id 查询用户数据信息
        UserDao userDao=(UserDao)applicationContext.getBean("userDao");
        User user=userDao.findUserById(2);
        System.out.println(user);
    }
```

上述代码通过执行 findUserById()方法获取了 id 为 1 的对象信息，并通过输出语句输出。使用 JUnit4 测试运行后，控制台的输出结果如图 4.9 所示。

```
Markers  Properties  Servers  Snippets  Problems  Console  JUnit  Terminal
<terminated> JdbcTemplateTest.findUserByIdTest [JUnit] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年10月24日 下午11:09:18)
十月 24, 2018 11:09:19 下午 org.springframework.jdbc.datasource.DriverManage
信息: Loaded JDBC driver: com.mysql.jdbc.Driver
User [id=2, username=lisi, password=666666]
```

图 4.9　运行结果

步骤05 在测试类 JdbcTemplateTest 中添加一个测试方法 findAllUserTest()来测试所有用户信息，其代码如下所示。

```
@Test
    public void findAllUserTest(){
        ApplicationContext applicationContext=
                new ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao=(UserDao)applicationContext.getBean("userDao");
    //查询所有用户数据信息
        List<User> list=userDao.findAllUser();
    //循环输出用户数据信息
        for(User user:list){
            System.out.println(user);
        }
    }
```

在上述代码中，调用了 UserDao 对象的 findAllUser()方法查询所有用户账户信息，并通过 for 循环输出查询结果。

使用 JUnit4 成功运行 findAllUser()方法后，控制台的显示信息如图 4.10 所示。从中可以看出，数据表 user 中的 4 条记录都已经被查询出来。

```
Markers  Properties  Servers  Snippets  Problems  Console  JUnit  Terminal
<terminated> JdbcTemplateTest.findAllUserTest [JUnit] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年10月24日 下午11:14:15)
十月 24, 2018 11:14:16 下午 org.springframework.jdbc.datasource.DriverManage
信息: Loaded JDBC driver: com.mysql.jdbc.Driver
User [id=2, username=lisi, password=666666]
User [id=3, username=zhangsan, password=123456]
User [id=4, username=wangwu, password=111111]
User [id=5, username=wujit, password=222222]
```

图 4.10　运行结果

4.3 习　　题

1. 请简述 Spring JDBC 是如何进行配置的。
2. 请简述 Spring JdbcTemplate 类中几个常用方法的作用。

第 5 章

Spring 的事务管理

第 4 章介绍了如何使用 Spring 对数据库进行基本操作，操作数据库时还涉及事务管理问题，为此 Spring 提供了专门用于事务处理的 API。本章将详细讲解 Spring 的事务管理功能。

本章主要涉及的知识点如下：

- Spring 事务管理概述：事务管理的核心接口和事务管理的方式。
- 声明式事务管理：基于 XML 方式的声明式事务和基于 Annotation 方式的声明式 事务。

5.1　Spring 事务管理概述

Spring 的事务管理简化了传统的事务管理流程，并且在一定程度上减少了开发者的工作量。

5.1.1　事务管理的核心接口

在 Spring 的所有 JAR 包中包含一个名为 Spring-tx-4.3.6.RELEASE 的 JAR 包，该包就是 Spring 提供的用于事务管理的依赖包。在该 JAR 包的 org.Springframework.transaction 包中有 3 个接口文件：PlatformTransactionManager、TransactionDefinition 和 TransactionStatus。

1. PlatformTransactionManager

PlatformTransactionManager 接口是 Spring 提供的平台事务管理器，主要用于管理事务。该接口中提供了 3 个事务操作的方法，具体如下。

- TransactionStatus getTransaction(TransactionDefinition definition)：用于获取事务状态信息。该方法会根据 TransactionDefinition 参数返回一个 TransactionStatus 对象。TransactionStatus 对

象表示一个事务，被关联在当前执行的线程上。

- void commit(TransactionStatus status)：用于提交事务。
- void rollback(TransactionStatus status)：用于回滚事务。

PlatformTransactionManager 接口只是代表事务管理的接口，并不知道底层是如何管理事务的，它只需要事务管理提供上面的 3 个方法，但具体如何管理事务则由它的实现类来完成。

PlatformTransactionManager 接口有许多不同的实现类，常见的几个实现类如下。

- org.springframework.jdbc.datasource.DataSourceTransactionManager：用于配置 JDBC 数据源的事务管理器。
- org.springframework.orm.Hibernate4.HibernateTransactionManager：用于配置 Hibernate 的事务管理器。
- org.springframework.transaction.jta.JtaTransactionManager：用于配置全局事务管理器。

当底层采用不同的持久层技术时，系统只需使用不同的 PlatformTransactionManager 实现类即可。

2. TransactionDefinition

TransactionDefinition 接口是事务定义（描述）的对象，该对象中定义了事务规则，并提供了获取事务相关信息的方法，具体如下：

- string getName()：获取事务对象名称。
- int getlsolationLeve()：获取事务的隔离级别。
- int getPropagationBehavior()：获取事务的传播行为。
- int setTimeout()：获取事务的超时时间。
- boolean isReadOnly()：获取事务是否只读。

上述方法中，事务的传播行为是指在同一个方法中，不同操作前后所使用的事务。传播行为有很多种，具体如表 5.1 所示。

表 5.1　传播行为的类

属性名称	值	描述
PROPAGATION_REQUIRED	REQUIRED	表示当前方法必须运行在一个事务环境中，如果当前方法已处于事务环境中，就可以直接使用该方法；否则会开启一个新事务后执行该方法
PROPAGATION_SUPPORTS	SUPPORTS	如果当前方法处于事务环境中，就使用当前事务，否则不使用事务
PROPAGATION_MANDATORY	MANDATORY	表示调用该方法的线程必须处于当前事务环境中，否则将抛出异常
PROPAGATION_REQUIRES_NEW	REQUIRES_NEW	要求方法在新的事务环境中执行。如果当前方法已在事务环境中，就先暂停当前事务，在启动新的事务后执行该方法；如果当前方法不在事务环境中，就会启动一个新的事务后执行该方法

（续表）

属性名称	值	描述
PROPAGATION_NOT_SUPPORTED	NOT_SUPPORTED	不支持当前事务，总是以非事务状态执行。如果调用该方法的线程处于事务环境中，就先暂停当前事务，然后执行该方法
PROPAGATION_NEVER	NEVER	不支持当前事务。如果调用该方法的线程处于事务环境中，将抛出异常
PROPAGATION_NESTED	NESTED	即使当前执行的方法处于事务环境中，依然会启动一个新的事务，并且方法在嵌套的事务里执行；即使当前执行的方法不在事务环境中，也会启动一个新事务，然后执行该方法

在事务管理过程中，传播行为可以控制是否需要创建事务以及如何创建事务。通常情况下，数据的查询不会影响原数据的改变，所以不需要进行事务管理，而对于数据的插入、更新和删除操作，必须进行事务管理。如果没有指定事务的传播行为，Spring 默认传播行为是 REQUIRED。

3. TransactionStatus

TransactionStatus 接口是事务的状态，描述了某一时间点上事务的状态信息。该接口中包含 6 个方法，具体如下：

- void flush()：刷新事务。
- boolean hasSavepoint()：获取是否存在保存点。
- boolean isCompleted()：获取事务是否完成。
- boolean isNewTransaction()：获取是否是新事务。
- boolean isRollbackOnly()：获取是否回滚。
- void setRollbackOnly()：设置事务回滚。

5.1.2　事务管理的方式

Spring 中的事务管理分为两种方式：一种是传统的编程序事务管理；另一种是声明式事务管理。

- 编程序事务管理：通过编写代码实现的事务管理，包括定义事务的开始、正常执行后的事务提交和异常时的事务回滚。
- 声明式事务管理：通过 AOP 技术实现的事务管理，其主要思想是将事务管理作为一个“切面”代码单独编写，然后通过 AOP 技术将事务管理的“切面”代码植入业务目标类中。

声明式事务管理最大的优点在于开发者无须通过编程的方式来管理事务，只需在配置文件中进行相关的事务规则声明，就可以将事务规则应用到业务逻辑中。这使得开发人员可以更加专注于核心业务逻辑代码的编写，在一定程度上减少了工作量，提高了开发效率。所以在实际开发中，通常都推荐使用声明式事务管理。

5.2 声明式事务管理

Spring 的声明式事务管理可以通过两种方式来实现：一种是基于 XML 的方式；另一种是基于 Annotation 的方式。本节将对这两种声明式事务管理方式进行详细讲解。

5.2.1 基于 XML 方式的声明式事务

基于 XML 方式的声明式事务管理是通过在配置文件中配置事务规则的相关声明来实现的。

Spring 2.0 以后，提供了 tx 命名空间来配置事务，tx 命名空间下提供了<tx:advice>元素来配置事务的通知（增强处理）。当使用<tx:advice>元素配置了事务的增强处理后，就可以通过编写的 AOP 配置让 Spring 自动对目标生成代理。

配置<tx:advice>元素时，通常需要指定 id 和 transaction-manager 属性，其中 id 属性是配置文件中的唯一标识，transaction-manager 属性用于指定事务管理器。除此之外，还需要配置一个<tx:attributes>子元素，该子元素可通过配置多个<tx:method>子元素来配置执行事务的细节。

关于<tx:method>元素的属性描述如表 5.2 所示。

表 5.2 <tx:method>元素的属性

属性名称	描述
name	必选属性，指定了与事务属性相关的方法名。其属性值支持使用通配符，如'*'、'get*'、'handle*'、'* Order'等
propagation	用于指定事务的传播行为，其属性值就是表 5.1 中的值，默认值为 REQUIRED
isolation	用于指定事务的隔离级别，其属性值可以为 DEFAULT、READ_UNCOMMITTED、READ_COMMITTED、REPEATABLE_READ 和 SERIALIZABLE，其默认值为 DEFAULT
read-only	用于指定事务是否只读，其默认值为 false
timeout	用于指定事务超时的时间，其默认值为-1，即永不超时
rollback-for	用于指定触发事务回滚的异常类，在指定多个异常类时，异常类之间以英文逗号分隔
no-rollback-for	用于指定不触发事务回滚的异常类，在指定多个异常类时，异常类之间以英文逗号分隔

了解了如何在 XML 文件中配置事务后，接下来通过一个案例来演示如何通过 XML 方式实现 Spring 的声明式事务管理。

【示例 5-1】本案例以第 4 章的项目代码和数据表为基础模拟一个会员赠送积分的功能，要求在赠送积分时通过 Spring 对事务进行控制，其具体实现步骤如下。

步骤01 在 Eclipse 中创建一个名为 chapter05 的 Web 项目，在项目的 lib 目录中导入 chapter04 项目中的所有 JAR 包，并将 AOP 所需 JAR 包也导入 lib 目录中。导入后的目录如图 5.1 所示。

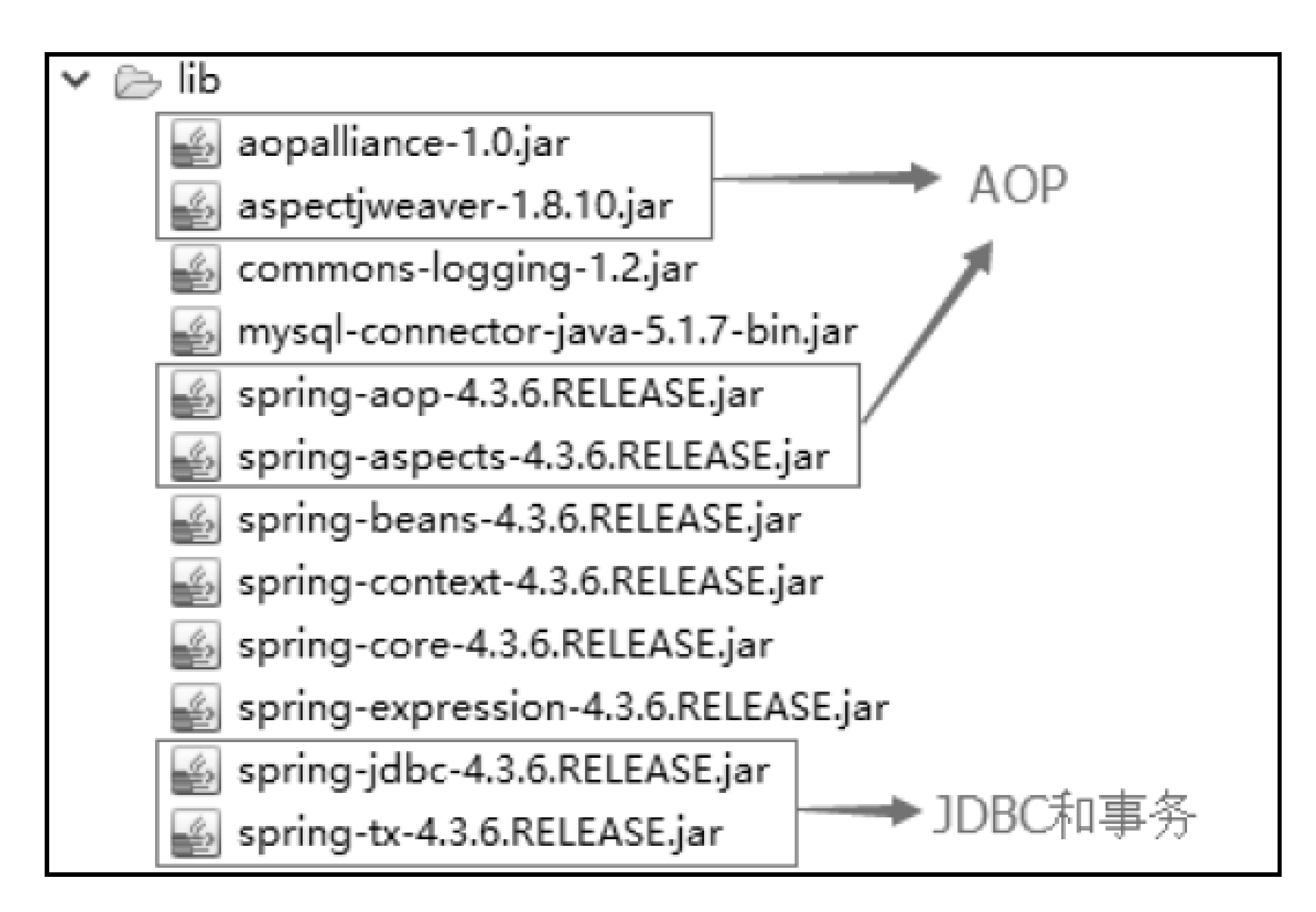


图 5.1　项目所需的 JAR 包

步骤02 在 MySQL 中，修改数据库 db_spring 中的数据表 user，增加字段 jf（积分），如图 5.2 所示。同时，为了后续操作数据表方便，在数据表 user 中设置所有用户的初始积分为 1000，如图 5.3 所示。

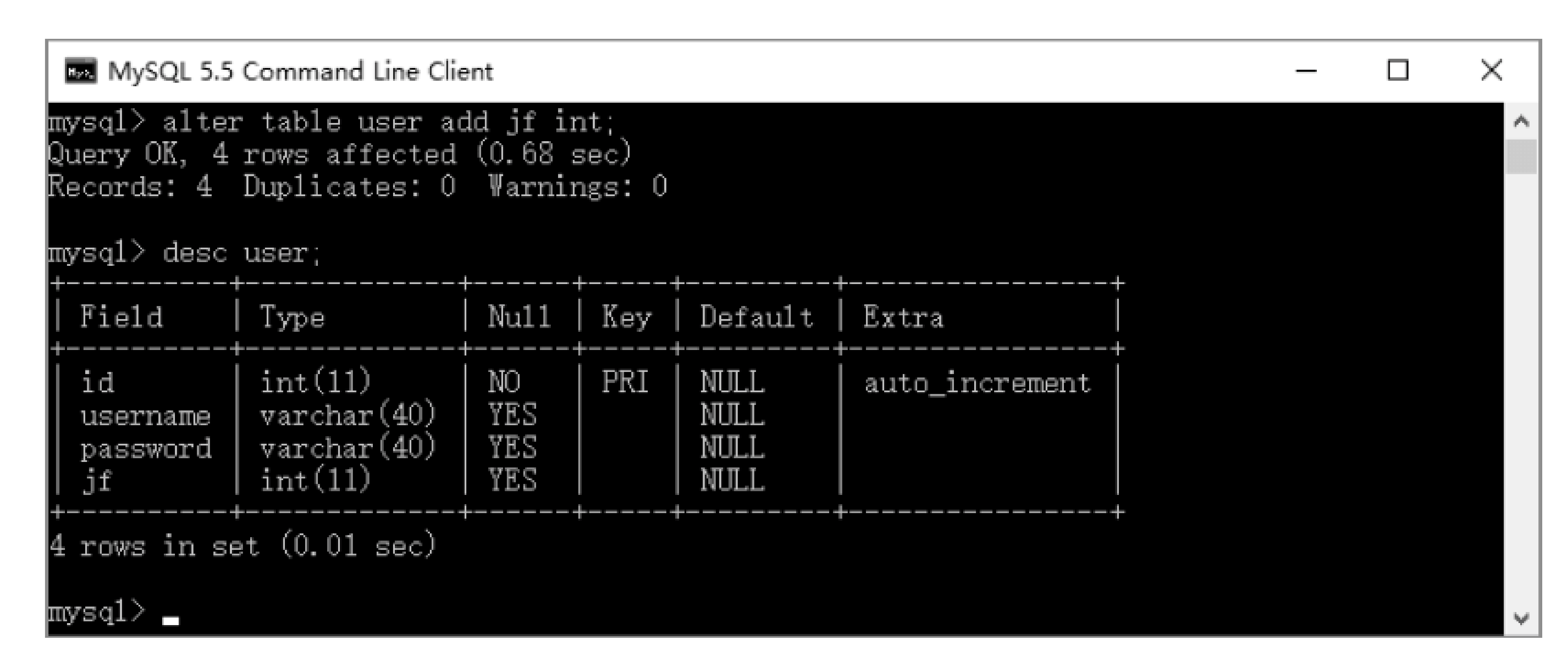


图 5.2　修改数据表

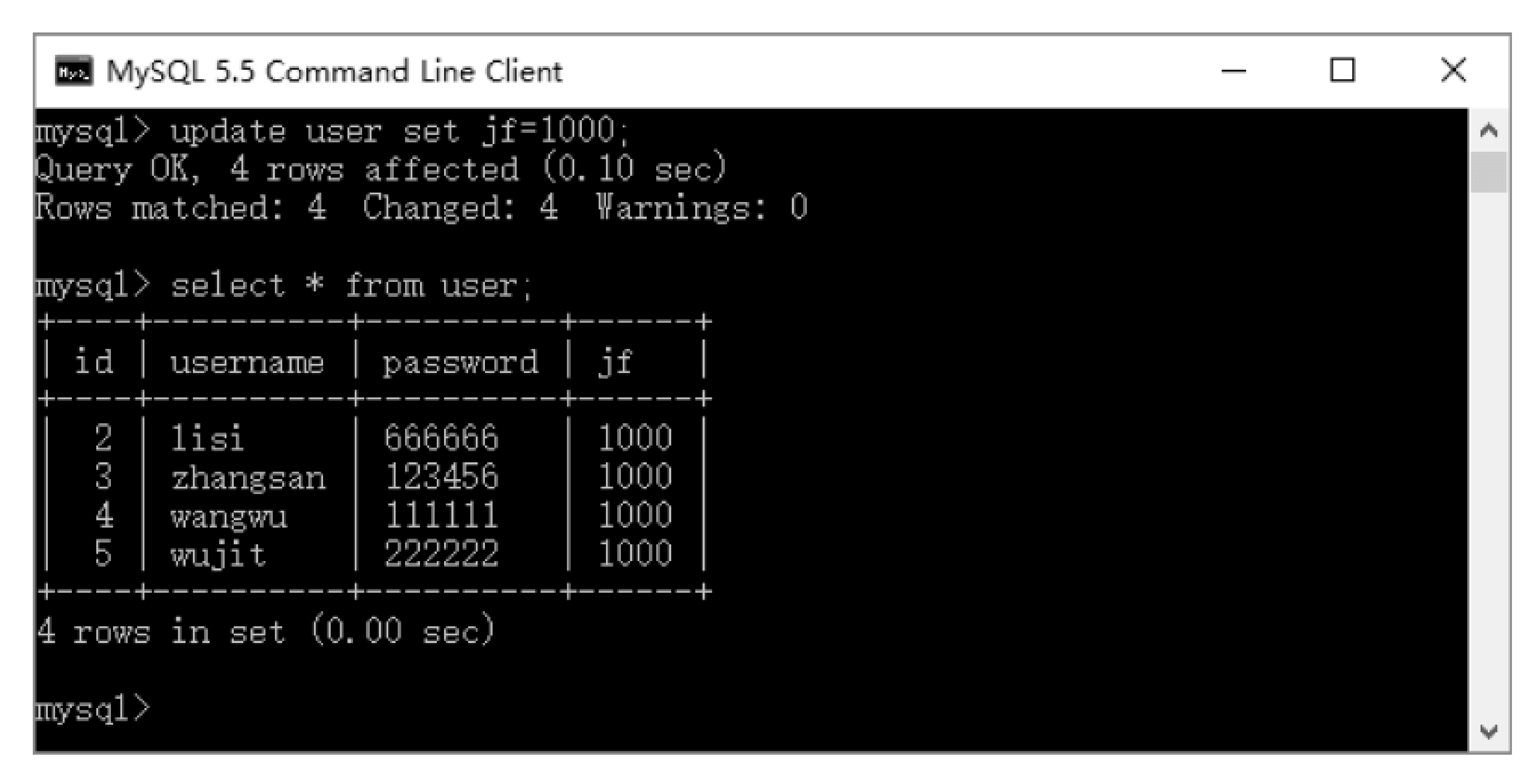


图 5.3　设置 jf（积分）初始值

步骤03 将 chapter04 项目中的代码和配置文件复制到 chapter05 项目的 src 目录下。接下来，在 User 类中增加 jf（积分）成员和对应的 getter()和 setter()方法，如下所示。

```
private Integer jf;
```

```
    public Integer getJf() {
        return jf;
    }
    public void setJf(Integer jf) {
        this.jf = jf;
    }
```

在 UserDao 接口中创建一个赠送积分的方法 transfer()，其代码如下所示。

```
    //赠送积分
    public void transfer(String outUser, String inUser, Integer jf);
```

在实现类 UserDaoImpl 中实现 transfer()方法，编辑后的代码如下所示。

```
    //赠送积分
    public void transfer(String outUser, String inUser, Integer jf) {
    //接收积分
    this.jdbcTemplate.update("update user set jf=jf+? where username=?",jf,inUser);
    //模拟系统运行时的突发性问题
    int i=1/0;
    //赠送（送出）积分
    this.jdbcTemplate.update("update user set jf =jf-? where username=? ", jf, outUser);
}
```

在上述代码中，使用了两个 update()方法对 user 表中的数据执行接收积分和赠送积分的更新操作。在两个操作之间添加了一行代码“int i=1/0;”来模拟系统运行时的突发性问题。如果没有事务控制，那么在 transfer()方法执行后，接收积分用户的积分会增加，而赠送积分用户的积分会因为系统出现问题而不变，这显然是有问题的；如果增加了事务控制，那么在 transfer()方法操作执行后，接收积分用户的积分和赠送积分用户的积分在问题出现前后都应该保持不变。

步骤04 修改配置文件 applicationContext.xml，添加命名空间并编写事务管理的相关配置代码，如文件 5.1 所示。

文件 5.1 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
```

```
    <!--1.配置数据源 -->
    <bean id="dataSource"
     class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <!--数据库驱动 -->
     <property name="driverClassName" value="com.mysql.jdbc.Driver" />
     <!--连接数据库的 url -->
     <property name="url" value="jdbc:mysql://localhost:3306/db_spring" />
     <!--连接数据库的用户名 -->
     <property name="username" value="root" />
     <!--连接数据库的密码 -->
     <property name="password" value="root" />
    </bean>
    <!--2.配置 JDBC 模板 -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
     <!--默认必须使用数据源 -->
     <property name="dataSource" ref="dataSource" />
    </bean>
    <!--3.定义 id 为 userDao 的 Bean -->
    <bean id="userDao" class="com.ssm.jdbc.UserDaoImpl">
     <!--将 jdbcTemplate 注入 userDao 实例中 -->
     <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
    <!--4.事务管理器，依赖于数据源 -->
    <bean id="transactionManager"
       class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
     <property name="dataSource" ref="dataSource"/>
    </bean>
    <!--5.编写通知：对事务进行增强（通知），需要编写对切入点和具体执行事务细节 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
     <tx:attributes>
         <tx:method name="*" propagation="REQUIRED"
           isolation="DEFAULT" read-only="false"/>
     </tx:attributes>
    </tx:advice>
    <!--6.编写 aop,让 spring 自动对目标生成代理，需要使用 AspectJ 的表达式 -->
    <aop:config>
     <!--切入点 -->
     <aop:pointcut expression="execution(* com.ssm.jdbc.*.*(..))" id="txPointCut" />
     <!--切面 -->
     <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
    </aop:config>
</beans>
```

在文件 5.1 中定义了 id 为 transactionManager 的事务管理器，接下来通过编写的通知来声明事务，最后通过声明 AOP 的方式让 Spring 自动生成代理。

步骤05 在 com.ssm.jdbc 包中创建测试类 TransactionTest，并在类中编写测试方法 xmlTest()，

如文件 5.2 所示。

文件 5.2 TransactionTest.java

```java
package com.ssm.jdbc;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class TransactionTest {
    @Test
    public void xmlTest(){
     ApplicationContext applicationContext =
                    new ClassPathXmlApplicationContext("applicationContext.xml");
     UserDao userDao=(UserDao)applicationContext.getBean("userDao");
     //赠送积分
     userDao.transfer("zhangsan","lisi", 100);
     System.out.println("赠送积分成功！");
    }
}
```

在文件 5.2 中获取 UserDao 实例后，调用了实例中的赠送积分方法，由 zhangsan 向 lisi 转入 100 积分。如果事务代码起作用，那么在整个赠送积分方法执行完毕后，zhangsan 和 lisi 的积分应该都是原来的值。

执行完文件 5.2 中的测试方法后，JUnit 控制台的显示结果如图 5.4 所示。从中可以看到，JUnit 控制台中报出了“/ y zero”的算术异常信息。

在执行赠送积分操作后，查看 user 表中的数据，依然如图 5.3 所示，即 zhangsan 和 lisi 的积分没有发生变化，这说明 Spring 中的事务管理配置已经生效。

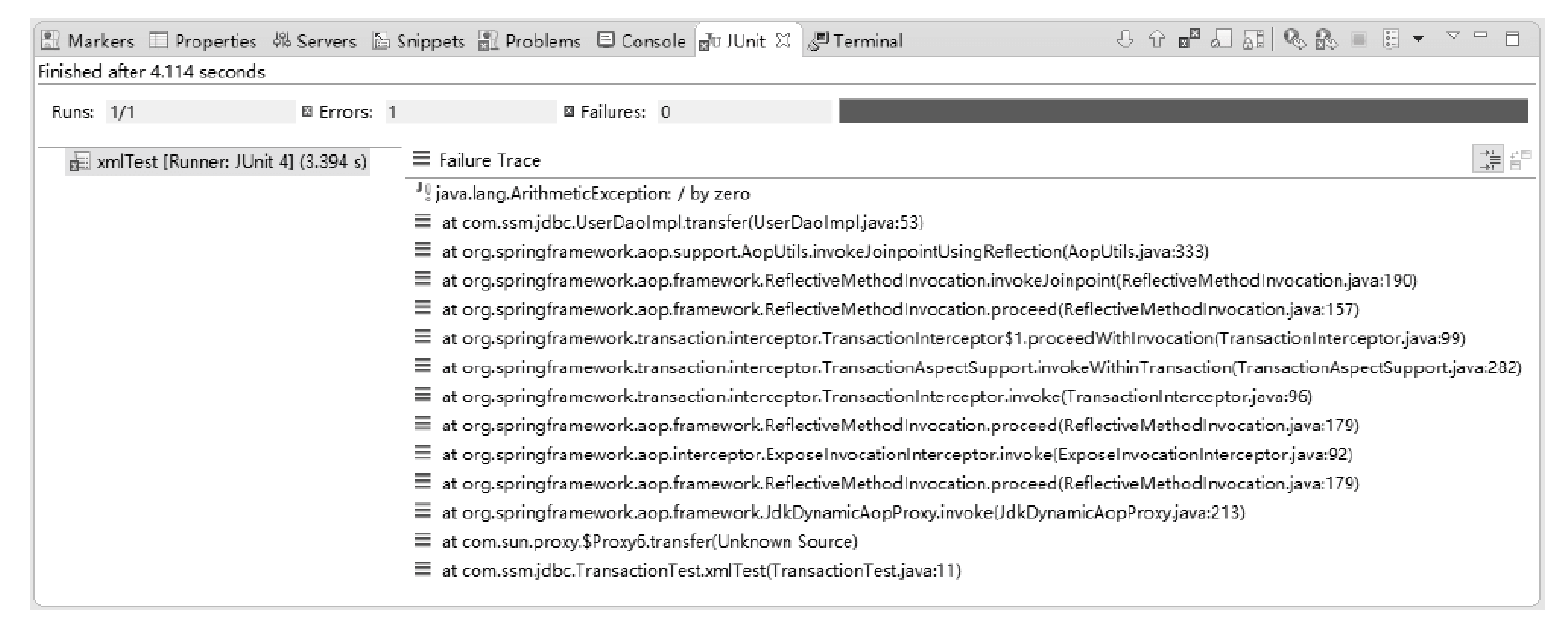


图 5.4 运行结果

5.2.2 基于 Annotation 方式的声明式事务

Spring 的声明式事务管理还可以通过 Annotation（注解）的方式来实现。这种方式的使用非常简单，开发者只需做两件事情：

（1）在 Spring 容器中注册事务注解驱动，其代码如下：

```
<tx:annotation-driven transaction-managers ntransactionManager"/>
```

（2）在需要使用事务的 Spring Bean 类或者 Bean 类的方法上添加注解@Transactional。如果将注解添加在 Bean 类上，就表示事务的设置对整个 Bean 类的所有方法都起作用；如果将注解添加在 Bean 类中的某个方法上，就表示事务的设置只对该方法有效。

使用@Transactional 注解时，可以通过其参数配置事务详情。@Transactional 注解可配置的参数信息如表 5.3 所示。

表 5.3　@Transactional 注解的参数及其描述

参数名称	描述
value	用于指定需要使用的事务管理器，默认为""，其别名为 transactionManager
transactionManager	指定事务的限定符值，可用于确定目标事务管理器，匹配特定的限定值（或者 Bean 的 name 值），默认为""，其别名为 value
isolation	用于指定事务的隔离级别，默认为 Isolation.DEFAULT（底层事务的隔离级别）
noRollbackFor	用于指定遇到特定异常时强制不回滚事务
noRolbackForClassName	用于指定遇到特定的多个异常时强制不回滚事务。其属性值可以指定多个异常类名
propagation	用于指定事务的传播行为，默认为 Propagation REQUIRED
read-only	用于指定事务是否只读，默认为 false
rollbackFor	用于指定遇到特定异常时强制回滚事务
rollbackForClassName	用于指定遇到特定的多个异常时强制回滚事务。其属性值可以指定多个异常类名
timeout	用于指定事务的超时时长，默认为 TransactionDefinition.TIMEOUT_DEFAULT（底层事务系统的默认时间）

从表 5.3 可以看出，@Transactional 注解与<tx:method>元素中的事务属性基本是对应的，并且其含义也基本相似。

【示例 5-2】为了让读者更加清楚地掌握@Transactiona 注解的使用，接下来对 5.2.1 小节的案例进行修改，以 Annotation 方式实现项目中的事务管理，具体实现步骤如下。

步骤01 在 src 目录下创建一个 Spring 配置文件 applicationContext-annotation.xml，在该文件中声明事务管理器等配置信息，如文件 5.3 所示。

文件 5.3　applicationContext-annotation.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
```

```xml
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.3.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!--1.配置数据源 -->
    <bean id="dataSource"
     class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <!--数据库驱动 -->
     <property name="driverClassName" value="com.mysql.jdbc.Driver" />
     <!--连接数据库的 url -->
     <property name="url" value="jdbc:mysql://localhost:3306/db_spring" />
     <!--连接数据库的用户名 -->
     <property name="username" value="root" />
     <!--连接数据库的密码 -->
     <property name="password" value="root" />
    </bean>
    <!--2.配置 JDBC 模板 -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
     <!--默认必须使用数据源 -->
     <property name="dataSource" ref="dataSource" />
    </bean>
    <!--3.定义 id 为 userDao 的 Bean -->
    <bean id="userDao" class="com.ssm.jdbc.UserDaoImpl">
     <!--将 jdbcTemplate 注入 userDao 实例中 -->
     <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
    <!--4.事务管理器，依赖于数据源 -->
    <bean id="transactionManager"
       class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
     <property name="dataSource" ref="dataSource"/>
    </bean>
    <!--5.注册事务管理器驱动 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

与基于 XML 方式的配置文件相比，文件 5.3 通过注册事务管理器驱动替换了文件 5.1 中的编写通知和编写 aop 的配置，这样大大减少了配置文件中的代码量。

注　意

如果案例中使用了注解式开发，就需要在配置文件中开启注解处理器，指定扫描哪些包下的注解。这里没有开启注解处理器是因为在配置文件中已经配置了 UserDaoImpl 类的 Bean，而@Transactional 注解就配置在该 Bean 类中，所以可以直接生效。

步骤02 在 UserDaoImpl 类的 transfer()方法上添加事务注解，添加后的代码如下所示。

```
@Transactional(propagation=Propagation.REQUIRED,isolation=Isolation.DEFAULT,
  readOnly=false)
    public void transfer(String outUser, String inUser, Integer jf) {
        //接收积分
        this.jdbcTemplate.update("update user set jf=jf+? where username=?",jf,inUser);
        //模拟系统运行时的突发性问题
        int i=1/0;
        //赠送（送出）积分
        this.jdbcTemplate.update("update user set jf =jf-? where username=? ", jf, outUser);
    }
```

上述方法已经添加了@Transactional 注解，并且使用注解的参数配置了事务详情，各个参数之间要用英文逗号“,”进行分隔。

注　意
在实际开发中，事务的配置信息通常是在 Spring 的配置文件中完成的，而在业务层类上只需使用@Transactional 注解即可，不需要配置@Transactional 注解的属性。

步骤03 在 TransactionTest 类中创建测试方法 annotationTest()，编辑后的代码如下所示。

```
@Test
public void annotationTest(){
    ApplicationContext applicationContext =
       new ClassPathXmlApplicationContext("applicationContext-annotation.xml");
    UserDao userDao=(UserDao)applicationContext.getBean("userDao");
   //赠送积分
    userDao.transfer("zhangsan","lisi", 200);
    System.out.println("赠送积分成功！");
}
```

从上述代码可以看出，与 XML 方式的测试方法相比，该方法只是对配置文件的名称进行了修改。程序执行后，会出现与 XML 方式同样的执行结果。

5.3　习　　题

1. 请简述 Spring 中事务管理的两种方式。
2. 请简述如何使用 Annotation 方式进行声明式事务管理。

第 6 章

初识 MyBatis

MyBatis 是当前主流的 Java 持久层框架之一，与 Hibernate 一样，也是一种 ORM 框架。因其性能优异，且具有高度的灵活性、可优化性和易于维护等特点，所以受到了广大互联网企业的青睐，是目前大型互联网项目的首选框架。接下来的几章将详细讲解 MyBatis 框架的相关知识。

本章主要涉及的知识点如下：

- MyBatis 概述：什么是 MyBatis 及其下载与使用。
- MyBatis 入门程序：查询用户、添加用户、更新用户、删除用户。

6.1 MyBatis 概述

MyBatis 是一个支持普通 SQL 查询、存储过程以及高级映射的持久层框架，它消除了几乎所有的 JDBC 代码和参数的手动设置以及对结果集的检索，并使用简单的 XML 或注解进行配置和原始映射，用以将接口和 Java 的 POJO（Plain Old Java Object，普通 Java 对象）映射成数据库中的记录，使得 Java 开发人员可以使用面向对象的编程思想来操作数据库。

6.1.1 什么是 MyBatis

MyBatis 框架也被称为 ORM（Object/ Relational Mapping，对象关系映射）框架。所谓 ORM，就是一种为了解决面向对象与关系型数据库中数据类型不匹配的技术，通过描述 Java 对象与数据库表之间的映射关系自动将 Java 应用程序中的对象持久化到关系型数据库的表中。

使用 ORM 框架后，应用程序不再直接访问底层数据库，而是以面向对象的方式来操作持久化对象（Persisent Object，PO），而 ORM 框架则会通过映射关系将这些面向对象的操作转换成底层的

SQL 操作。

当前的 ORM 框架产品有很多，常见的 ORM 框架有 Hibernate 和 MyBatis。

（1）Hibernate：一个全表映射的框架。通常开发者只需定义好持久化对象到数据库表的映射关系，就可以通过 Hibernate 提供的方法完成持久层操作。开发者并不需要熟练地掌握 SQL 语句的编写，Hibernate 会根据制定的存储逻辑自动生成对应的 SQL，并调用 JDBC 接口来执行，所以其开发效率会高于 MyBatis。然而 Hibernate 自身存在着一些缺点，例如它在多表关联时，对 SQL 查询的支持较差；更新数据时，需要发送所有字段；不支持存储过程；不能通过优化 SQL 来优化性能等。这些问题导致其只适合在场景不太复杂且对性能要求不高的项目中使用。

（2）MyBatis：一个半自动映射的框架。这里所谓的“半自动”是相对于 Hibernate 全表映射而言的，MyBatis 需要手动匹配提供 POJO、SQL 和映射关系，而 Hibernate 只需提供 POJO 和映射关系即可。与 Hibernate 相比，虽然使用 MyBatis 手动编写 SQL 要比使用 Hibernate 的工作量大，但 MyBatis 可以配置动态 SQL 并优化 SQL，可以通过配置决定 SQL 的映射规则，它还支持存储过程等。对于一些复杂的和需要优化性能的项目来说，显然使用 MyBatis 更加合适。

6.1.2　MyBatis 的下载和使用

MyBatis 可以到官网上下载。本书使用的版本是 mybatis-3.4.2，可以通过网址“https://github.com/mybatis/mybatis-3/releases”下载得到。下载并解压 mybatis-3.4.2.zip 压缩包，会得到一个名为 mybatis-3.4.2 的文件夹，里面有核心包 mybatis-3.4.2.jar 和一个 lib 文件夹（里面有 MyBatis 的依赖包）。

使用 MyBatis 框架非常简单，只需在应用程序中引入 MyBatis 的核心包 mybatis-3.4.2.jar 和 lib 目录中的依赖包即可。

注　意
如果底层采用的是 MySQL 数据库，还需要将 MySQL 数据库的驱动 JAR 包添加到应用程序的类路径中；如果采用其他类型的数据库，同样需要将对应类型的数据库驱动包添加到应用程序的类路径中。

6.2　MyBatis 入门程序

6.1 节对 MyBatis 框架有了一个初步的介绍。本节将通过一个用户信息管理的入门案例来讲解 MyBatis 框架的基本使用。

6.2.1　查询用户

在实际开发中，查询操作通常都会涉及单条数据的精确查询以及多条数据的模糊查询。那么使用 MyBatis 框架是如何进行这两种查询的呢？接下来，本小节将讲解如何使用 MyBatis 框架根据用

户编号查询客户信息，以及根据用户名模糊查询用户信息。

1. 根据用户编号（id）查询用户信息

【示例 6-1】根据用户编号查询用户信息主要是通过查询用户表中的主键（这里表示唯一的用户编号）来实现的，其具体实现步骤如下。

步骤01 在 MSQL 数据库中创建一个名为 db_mybatis 的数据库，在此数据库中创建一个 t_user 表，同时预先添加几条数据，对应的 SQL 语句如下所示。

```
#创建数据库 db_mybatis
CREATE DATABASE db_mybatis;
//使用数据库 db_mybatis
USE db_mybatis;
#创建数据表 t_user
CREATE TABLE t_user(
    id int(32) PRIMARY KEY AUTO_INCREMENT,
    username varchar(50),
    jobs varchar(50),
    phone varchar (16)
);
#向数据表 t_user 中插入数据记录
INSERT INTO t_user VALUES(1,'zhangsan','teacher','13907998372');
INSERT INTO t_user VALUES(2,'lisi','worker','13907396542');
INSERT INTO t_user VALUES(3,'wangwu','doctor','13817348729');
```

执行上述代码后，t_user 表中的数据如图 6.1 所示。

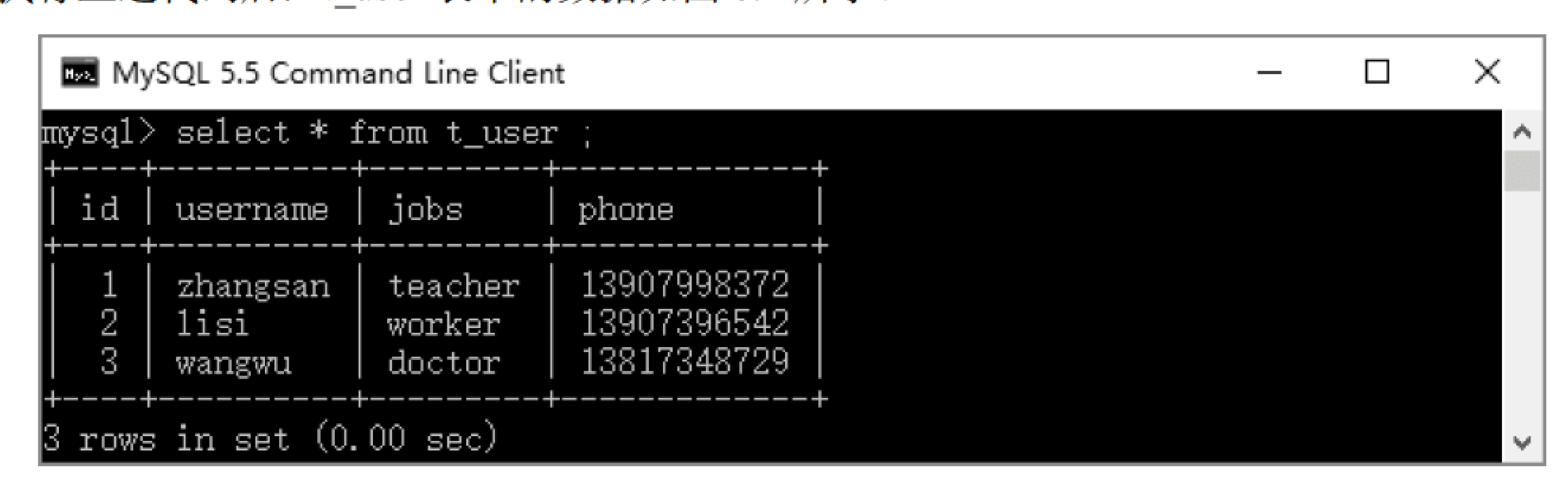


图 6.1 t_user 表数据

步骤02 在 Eclipse 中创建一个名为 chapter06 的 Web 项目，将 MyBatis 的核心 JAR 包、lib 目录中的依赖 JAR 包以及 MySQL 数据库的驱动 JAR 包一同添加到项目的 lib 目录下，并发布到类路径中。添加后的 lib 目录如图 6.2 所示。



图 6.2　MyBatis 相关 JAR 包

步骤03 由于 MyBatis 默认使用 log4j 输出日志信息，因此如果要查看控制台的输出 SQL 语句，就需要在 classpath 路径下配置其日志文件。在项目的 src 目录下创建 log4j.properties 文件，内容如文件 6.1 所示。

文件 6.1　log4j.properties

```
# Global logging configuration
log4j.rootLogger=ERROR, stdout
# MyBatis logging configuration...
log4j.logger.com.ssm=DEBUG
Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

在文件 6-1 中包含全局的日志配置、MyBatis 的日志配置和控制台输出，其中 MyBatis 的日志配置用于将 com.ssm 包下所有类的日志记录级别设置为 DEBUG。

步骤04 在 src 目录下创建一个 com.ssm.po 包，在该包下创建持久化类 User，并在类中声明 id、username、jobs 和 phone 属性，及其对应的 getter/setter 法，如文件 6.2 所示。

文件 6.2　User.java

```
package com.ssm.po;
//用户类 User
public class User {
    private Integer id; //用户 id
    private String username; //用户姓名
    private String jobs;  //用户职业
    private String phone; //用户电话号码
    public Integer getId() {
    return id;
    }
    public void setId(Integer id) {
```

```
        this.id = id;
        }
        public String getUsername() {
        return username;
        }
        public void setUsername(String username) {
        this.username = username;
        }
        public String getJobs() {
        return jobs;
        }
        public void setJobs(String jobs) {
        this.jobs = jobs;
        }
        public String getPhone() {
        return phone;
        }
        public void setPhone(String phone) {
        this.phone = phone;
        }
        public String toString() {
        return "User [id=" + id + ", username=" + username + ", jobs=" + jobs +
         ", phone=" + phone + "]";
        }
    }
```

从上述代码可以看出，持久化类 User 与普通的 JavaBean 并没有什么区别，只是其属性字段与数据库中的表字段相对应。实际上，User 就是一个 POJO（普通 Java 对象）。MyBatis 就是采用 POJO 作为持久化类来完成对数据库操作的。

步骤05 在 src 目录下创建一个 com.ssm.mapper 包，并在包中创建映射文件 UserMapper.xml，内容如文件 6.3 所示。

文件 6.3 UserMapper.xml

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
03     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
04 <mapper namespace="com.ssm.mapper.UserMapper">
05     <!--根据用户编号获取用户信息 -->
06     <select id="findUserById" parameterType="Integer" resultType="com.ssm.po.User">
07      select * from t_user where id=#{id}
08     </select>
09 </mapper>
```

在文件 6.3 中，第 02、03 行是 MyBatis 的约束配置，第 04~09 行是需要程序员编写的映射信息。其中，<mapper>元素是配置文件的根元素，包含一个 namespace 属性，该属性为<mapper>元素指定了唯一的命名空间，通常会设置成“包名+SQL 映射文件名”的形式。子元素<select>中的信息是用

于执行查询操作的配置，其 id 属性是<select>元素在映射文件中的唯一标识；parameterType 属性用于指定传入参数的类型，这里表示传递给执行 SQL 的是一个 Integer 类型的参数；resultType 属性用于指定返回结果的类型，这里表示返回的数据是 Customer 类型。在定义的查询 SQL 语句中，“#{}”用于表示一个占位符，相当于“?”；而“#{id}”表示该占位符待接收参数的名称为 id。

步骤06 在 src 目录下创建 MyBatis 的核心配置文件 mybatis-config.xml，编辑后如文件 6.4 所示。

文件 6.4　mybatis-config.xml

```
01  <?xml version="1.0" encoding="UTF-8"?>
02  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
03      "http://mybatis.org/dtd/mybatis-3-config.dtd">
04  <configuration>
05  <!--1.配置环境，默认的环境 id 为 mysql -->
06      <environments default="mysql">
07    <!--1.2 配置 id 为 mysql 的数据库环境 -->
08       <environment id="mysql">
09           <!-- 使用 JDBC 的事务管理 -->
10           <transactionManager type="JDBC" />
11           <!-- 数据库连接池 -->
12           <dataSource type="POOLED">
13               <property name="driver" value="com.mysql.jdbc.Driver" />
14               <property name="url" value="jdbc:mysql://localhost:3306/db_mybatis" />
15               <property name="username" value="root" />
16               <property name="password" value="root" />
17           </dataSource>
18       </environment>
19      </environments>
20  <!—2.配置 Mapper 的位置 -->
21      <mappers>
22       <mapper resource="com/ssm/mapper/UserMapper.xml" />
23      </mappers>
24  </configuration>
```

在文件 6.4 中，第 02、03 行是 MyBatis 的配置文件的约束信息。下面<configuration>元素中的内容是开发人员需要编写的配置信息。这里按照<configuration>子元素的功能将配置分为了两个步骤：第 1 步，配置环境；第 2 步，配置 mapper 的位置。

步骤07 在 src 目录下创建一个 com.ssm.test 包，在该包下创建测试类 MybatisTest，并在类中编写测试方法 findUserByIdTest()，如文件 6.5 所示。

文件 6.5　MybatisTest.java

```
package com.ssm.test;
import java.io.InputStream;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
```

```
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Test;
import com.ssm.po.User;
/**
* 入门程序测试类
*/
public class MybatisTest {
 /**
 * 根据用户 id 查询用户信息
 */
@Test
public void findUserByIdTest() throws Exception{
  //1.读取配置文件
     String resource="mybatis-config.xml";
     InputStream inputStream=Resources.getResourceAsStream(resource);
  //2.根据配置文件构建 SqlSessionFactory 实例
     SqlSessionFactory sqlSessionFactory=new SqlSessionFactoryBuilder().build(inputStream);
  //3.通过 SqlSessionFactory 创建 SqlSession 实例
     SqlSession sqlSession=sqlSessionFactory.openSession();
  //4. SqlSession 执行映射文件中定义的 SQL，并返回映射结果
     User user=sqlSession.selectOne("com.ssm.mapper.UserMapper.findUserById",1);
  //5.打印输出结果
     System.out.println(user.toString());
  //6.关闭 SqlSession
     sqlSession.close();
     }
}
```

在文件 6.5 的 findUserByIdTest()方法中，首先通过输入流读取了配置文件，然后根据配置文件构建了 SqlSessionFactory 对象。接下来通过 SqlSessionFactory 对象又创建了 SqlSession 对象，并通过 SqlSession 对象的 selectOne()方法执行查询操作。selectOne()方法的第 1 个参数表示映射 SQL 的标识字符串，由 UserMapper.xml 中<mapper>元素的 namespace 属性值+<select>元素的 id 属性值组成；第 2 个参数表示查询所需要的参数，这里查询的是用户表中 id 为 1 的用户。为了查看查询结果，这里使用了输出语句输出查询结果信息。最后，程序执行完毕时，关闭了 SqlSession。

使用 JUnit4 测试执行 findUserByIdTest()方法后，控制台的输出结果如图 6.3 所示。

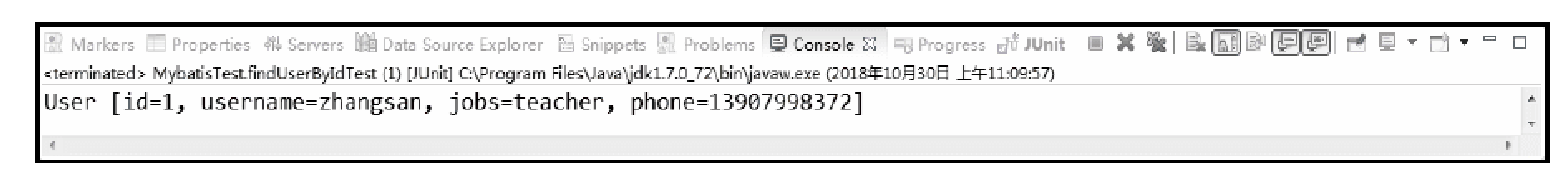


图 6.3 运行结果

2. 根据用户名模糊查询用户信息

接下来讲解如何根据用户名来模糊查询相关的用户信息。

【示例 6-2】模糊查询的实现只需要在映射文件中通过<select>元素编写相应的 SQL 语句，并通过 SqlSession 的查询方法执行该 SQL 语句即可。其具体实现步骤如下。

步骤01 在映射文件 UserMapper.xml 中添加根据用户名模糊查询用户信息列表的 SQL 语句，具体实现代码如下。

根据用户名模糊查询用户信息列表

```
<!--根据用户名模糊查询用户信息 -->
<select id="findUserByName" parameterType="String" resultType="com.ssm.po.User">
select * from t_user where username like '%${value}%'
</select>
```

与根据用户编号查询相比，上述配置代码中的属性 id、parameterType 和 SQL 语句都发生了相应变化。其中，SQL 语句中的“${}”用来表示拼接 SQL 的字符串，即不加解释地原样输出。“${value}”表示要拼接的是简单类型参数。

注　意

在使用 “${}” 进行 SOL 字符串拼接时，无法防止 SQL 注入问题。想要既能实现模糊查询，又能防止 SOL 注入，可以对上述映射文件 UserMapper.xml 中模糊查询的 select 语句进行修改，使用 MySQL 中的 concat 函数进行字符串拼接。具体修改示例如下所示。

```
select * from t_user where username like concat('%', ${value},'%')
```

步骤02 在测试类 MybatisTest 中添加一个测试方法 findUserByNameTest()，其代码如下所示。

```
@Test
public void findUserByNameTest() throws Exception{
        String resource = "mybatis-config.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory =
                    new SqlSessionFactoryBuilder().build(inputStream);
        SqlSession sqlSession = sqlSessionFactory.openSession();
        List<User> users =
           sqlSession.selectList("com.ssm.mapper.UserMapper.findUserByName", "g");
        //循环输出结果集
       for (User user : users) {
            System.out.println(user.toString());
        }
        sqlSession.close();
    }
```

在上述代码中，由于可能查询出多条数据，因此调用 SqlSession 的 selectList()方法来查询返回结果的集合对象，并使用 for 循环输出结果集对象。执行 findUserByNameTest ()方法后，控制台的输出结果如图 6.4 所示。

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Problems  Console  Progress  JUnit
<terminated> MybatisTest.findUserByNameTest [JUnit] C:\Program Files\Java\jdk1.7.0_72\bin\javaw.exe (2018年10月30日 下午1:50:41)
User [id=1, username=zhangsan, jobs=teacher, phone=13907998372]
User [id=3, username=wangwu, jobs=doctor, phone=13817348729]
```

图 6.4　运行结果

从图 6.4 可以看出，使用 MyBatis 框架已成功查询出了用户表中用户名带有“g”的两条客户信息。

至此，MyBatis 入门程序的查询功能就已经讲解完了。从上面两个查询方法中可以发现，MyBatis 的操作大致可分为以下几个步骤。

步骤01 读取配置文件。

步骤02 根据配置文件构建 SqlSessionFactory。

步骤03 通过 SqlSessionFactory 创建 SqlSession。

步骤04 使用 SqlSession 对象操作数据库（包括查询、添加、修改、删除以及提交事务等）。

步骤05 关闭 SqlSession。

6.2.2 添加客户

在 MyBatis 的映射文件中，添加操作是通过<insert>元素来实现的。例如，向数据库中的 t_user 表中插入一条数据可以通过如下配置来实现。

```
<!--添加用户信息 -->
<insert id="addUser" parameterType="com.ssm.po.User">
insert into  t_user(username,jobs,phone)values(#{username},#{jobs},#{phone})
</insert>
```

在上述配置代码中，传入的参数是一个 User 类型，该类型的参数对象被传递到语句中时，#{username}会查找参数对象 User 的 username 属性，#{jobs}和#{phone}也是一样的，并将其属性值传入 SQL 语句中。

【示例 6-3】为了验证上述配置是否正确，下面编写一个测试方法来执行添加操作。

在测试类 MybatisTest 中添加测试方法 addUserTest()，其代码如下所示。

```
@Test
public void addUserTest() throws Exception {
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory =
                new SqlSessionFactoryBuilder().build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
   //创建 User 对象，并向对象中添加数据
    User user = new User();
    user.setUsername("tom");
    user.setJobs("worker");
    user.setPhone("13624589654");
   //执行 SqlSession 的插入方法，返回 SQL 语句影响的行数
    int rows = sqlSession.insert("com.ssm.mapper.UserMapper.addUser", user);
   //通过返回结果判断插入操作是否执行成功
    if (rows > 0) {
        System.out.println("成功添加" + rows + "条数据！");
    } else {
```

```
        System.out.println("添加数据失败!");
    }
    //提交事务
    sqlSession.commit();
    //关闭 SqlSession
    sqlSession.close();
}
```

上述代码中，创建了 User 对象，并向 User 对象中添加了属性值；然后通过 SqlSession 对象的 insert()方法执行插入操作，并通过该操作返回的数据来判断插入操作是否执行成功；最后通过 SqlSession 的 commit()方法提交事务，并通过 SqlSession.close()方法关闭了 SqlSession。

6.2.3　更新用户

MyBatis 的更新操作在映射文件中是通过配置<update>元素来实现的。如果需要更新用户数据，可以通过如下代码配置来实现。

```
<!--更新用户信息 -->
<update id="updateUser" parameterType="com.ssm.po.User">
    update t_user set username=#{username},jobs=#{jobs},phone=#{phone} where id=#{id}
</update>
```

与插入数据的配置相比，更新操作配置中的元素与 SQL 语句都发生了相应变化，但其属性名却没有变。

【示例 6-4】为了验证配置是否正确，下面以 6.2.2 小节中新插入的数据为例进行更新用户测试。

在测试类 MybatisTest 中添加测试方法 updateUserTest()，将 id 为 4 的用户的 jobs 属性值修改为“teacher”，其代码如下所示。

```
@Test
public void updateUserTest() throws Exception {
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    //创建 User 对象，并对对象中的数据进行模拟更新
    User user = new User();
    user.setId(4);
    user.setUsername("tom");
    user.setJobs("teacher");
    user.setPhone("13624589654");
    //执行 SqlSession 的更新方法，返回 SQL 语句影响的行数
    int rows = sqlSession.update("com.ssm.mapper.UserMapper.addUser", user);
    if (rows > 0) {
        System.out.println("成功修改了" + rows + "条数据！");
    } else {
```

```
            System.out.println("修改数据失败!");
        }
        sqlSession.commit();
        sqlSession.close();
}
```

与添加用户的方法相比，更新操作的代码增加了 id 属性值的设置，并调用 SqlSession 的 update()方法对 id 为 4 的用户的 jobs 属性值进行修改。

6.2.4 删除用户

MyBatis 的删除操作在映射文件中是通过配置<delete>元素来实现的。在映射文件 UserMapper.xml 中添加删除客户信息的 SQL 语句，其示例代码如下。

```
<!--删除用户信息-->
<delete id="deleteUser" parameterType="Integer">
    delete from t_user where id=#{id}
</delete>
```

从上述配置的 SQL 语句中可以看出，我们只需要传递一个 id 值就可以将数据表中相应的数据删除。要测试删除操作的配置十分简单，只需使用 SqlSession 对象的 delete()方法传入需要删除数据的 id 值即可。

【示例 6-5】在测试类 MybatisTest 中添加测试方法 deleteUserTest()，该方法用于将 id 为 4 的用户信息删除，其代码如下所示。

```
@Test
public void deleteUserTest() throws Exception {
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
   //执行 SqlSession 的删除方法，返回 SQL 语句影响的行数
    int rows = sqlSession.delete("com.ssm.mapper.UserMapper.deleteUser", 4);
    if (rows > 0) {
        System.out.println("成功删除了" + rows + "条数据! ");
    } else {
        System.out.println("删除数据失败!");
    }
    sqlSession.commit();
    sqlSession.close();
}
```

至此，MyBatis 入门程序的增、删、改、查操作已经讲解完了。关于程序中映射文件和配置文件中的元素信息，将在第 7 章详细讲解，本章入门程序只需要了解所使用的元素即可。

6.3 习　　题

1. 请简述 MyBatis 框架及其与 Hibernate 框架的区别。
2. 上机调试一个具有增、删、改、查操作的 MyBatis 入门程序。

第 7 章

MyBatis 的核心配置

要想熟练地使用 MyBatis 框架进行实际开发，需要对框架中的核心对象以及映射文件和配置文件有深入的了解。本章将对这些内容进行详细的讲解。

本章主要涉及的知识点如下：

- MyBatis 核心对象。
- MyBatis 配置文件。
- MyBatis 映射文件。

7.1 MyBatis 的核心对象

MyBatis 框架主要涉及两个核心对象：SqlSessionFactory 和 SqlSession。本节将详细介绍这两个对象。

7.1.1 SqlSessionFactory

SqlSessionFactory 是单个数据库映射关系经过编译后的内存镜像，用于创建 SqlSession。SqlSessionFactory 对象的实例通过 SqlSessionFactoryBuilder 对象来构建，通过 XML 配置文件或一个预先定义好的 Configuration 实例构建出 SqlSessionFactory 的实例。通过 XML 配置文件构建出 SqlSessionFactory 实例的实现代码如下：

```
//读取配置文件
InputStream inputStream = Resources.getResourceAsStream("配置文件位置");
//根据配置文件构建 sqlSessionFactory
```

```
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

SqlSessionFactory 对象是线程安全的，一旦被创建，在整个应用执行期间都会存在。如果多次创建同一个数据库的 SqlSessionFactory，那么此数据库的资源将很容易被耗尽。所以在构建 SqlSessionFactory 实例时，建议使用单列模式。

7.1.2　SqlSession

SqlSession 是应用程序与持久层之间执行交互操作的一个单线程对象，其主要作用是执行持久化操作。SqlSession 对象包含数据库中所有执行 SQL 操作的方法，底层封装了 JDBC 连接，所以可以直接使用其实例来执行已映射的 SQL 语句。SqlSession 实例是不能被共享的，也是线程不安全的，因此其使用范围最好限定在一次请求或一个方法中，绝不能将其放在一个类的静态字段、实例字段或任何类型的管理范围中使用。使用完 SqlSession 对象之后，要及时将它关闭，通常可以将其放在 finally 块中关闭。

SqlSession 对象常用方法如下所示。

（1）<T> T selectOne(String statement);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id。该方法返回执行 SQL 语句查询结果的一个泛型对象。

（2）<T>I selectOne(String statement, Object parameter);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数。该方法返回执行 SQL 语句查询结果的一个泛型对象。

（3）<E> List<E> selectList(String statement);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id。该方法返回执行 SQL 语句查询结果的泛型对象的集合。

（4）<E> List<E> selectList(String statement, Object parameter);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数。该方法返回执行 SQL 语句查询结果的泛型对象的集合。

（5）e <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数，rowBounds 是用于分页的参数对象。该方法返回执行 SQL 语句查询结果的泛型对象的集合。

（6）void select(String statement, Object parameter, ResultHandler handler);

查询方法。参数 statement 是在配置文件中定义的<select>元素的 id，parameter 是查询所需的参数，ResultHandler 对象用于处理查询返回的复杂结果集，通常用于多表查询。

（7）int insert(String statement);

插入方法。参数 statement 是在配置文件中定义的<insert>元素的 id。该方法返回执行 SQL 语句所影响的行数。

（8）int insert(String statement, Object parameter);

插入方法。参数 statement 是在配置文件中定义的<insert>元素的 id，parameter 是插入所需的参数。该方法返回执行 SQL 语句所影响的行数。

（9）int update(String statement);

更新方法。参数 statement 是在配置文件中定义的<update>元素的 id。该方法返回执行 SQL 语句所影响的行数。

（10）int update(String statement, Object parameter);

更新方法。参数 statement 是在配置文件中定义的<update>元素的 id，parameter 是更新所需的参数。该方法返回执行 SQL 语句所影响的行数。

（11）int delete(String statement);

删除方法。参数 statement 是在配置文件中定义的<delete>元素的 id。该方法返回执行 SQL 语句所影响的行数。

（12）int delete(String statement, Object parameter);

删除方法。参数 statement 是在配置文件中定义的<delete>元素的 id，parameter 是删除所需的参数。该方法返回执行 SQL 语句所影响的行数。

（13）void commit();

提交事务的方法。

（14）void rollback();

回滚事务的方法。

（15）void close();

关闭 SqlSession 对象。

（16）<T>T getMapper(Class<T> type)

返回 Mapper 接口的代理对象，该对象关联了 Sqlsession 对象，开发人员可以使用该对象直接调用方法操作数据库。参数 type 是 Mapper 的接口类型。

（17）Connection getConnection();

获取 JDBC 数据库连接对象的方法。

注　意

为了简化开发，可以将构建 SqlSessionFactory 对象、创建 SqlSession 对象等重复性代码封装到一个工具类中，然后通过工具类来创建 SqlSession。

7.2　MyBatis 配置文件元素

使用 MyBatis 框架进行开发，需要创建 MyBatis 的核心配置文件，该配置文件包含重要的元素，熟悉配置文件中各个元素的功能十分重要。接下来的几个小节中，将对 MyBatis 配置文件中的主要元素进行详细的讲解。

在 MyBatis 框架的核心配置文件中，<configuration>元素是配置文件的根元素，其他元素都要在<contiguration>元素内配置。

MyBatis 配置文件中的主要元素如下所示。

```
<configuration>
```

```
    <!-- 属性 -->
    <properties/>
    <!-- 设置 -->
    <settings/>
    <!-- 类型命名 -->
    <typeAliases/>
    <!-- 类型处理器 -->
    <typeHandlers/>
    <!-- 对象工厂 -->
    <objectFactory/>
    <!-- 插件 -->
    <plugins/>
    <!-- 配置环境 -->
    <environments>
        <!-- 环境变量 -->
        <environment>
            <!-- 事务管理器 -->
            <transactionManager/>
            <!-- 数据源-->
            <dataSource/>
        </environment>
    </environments>
    <!-- 数据库厂商标识 -->
    <databaseIdProvider/>
    <!-- 映射器 -->
    <mappers/>
</configuration>
```

在 MyBatis 的配置文件中包含多个元素，这些元素在配置文件中分别发挥着不同的作用。开发人员需要熟悉的是<configuration>元素各个子元素的配置。

注　意
<configuration>的子元素必须按照由上到下的顺序进行配置，否则 MyBatis 在解析 XML 配置文件的时候会报错。

7.2.1　<properties>元素

<properties>是一个配置属性的元素，通过外部配置来动态替换内部定义的属性。

【示例 7-1】配置数据库的连接等属性，具体方式如下。

步骤01 在项目的 src 目录下创建一个名称为 db.properties 的配置文件，代码如下所示。

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql: //localhost: 3306/db_mybatis
jdbc.username=root
```

```
jdbc.password=root
```

步骤02 在 MyBatis 配置文件 mybatis-config.xml 中配置<properties/>属性，具体如下。

```
<properties resource="db.properties"/>
```

步骤03 修改配置文件中数据库连接的信息，具体如下。

```
<dataSource type="POOLED">
    <!--数据库驱动 -->
    <property name="driver" value="${jdbc.driver}" />
    <!--连接数据库的 url -->
    <property name="url" value="${jdbc.url}" />
    <!--连接数据库的用户名 -->
    <property name="username" value="${jdbc.username}" />
    <!--连接数据库的密码-->
    <property name="password" value="${jdbc.password}" />
</dataSource>
```

完成上述配置，dataSource 中连接数据库的 4 个属性（driver、url、username 和 password）值将会由 db.properties 文件中对应的值来动态替换。这样就为配置提供了灵活性。

另外，还可以通过配置<properties>元素的子元素<property>以及通过方法参数传递的方式来获取属性值。

由于使用 properties 配置文件来配置属性值可以方便地在多个配置文件中使用这些属性值，并且方便维护和修改，因此在实际开发中最常用。

7.2.2 <settings>元素

<settings>元素主要用于改变 MyBatis 运行时的行为，例如开启二级缓存、开启延迟加载等。<settings>元素中的常见配置及其描述如表 7.1 所示。

表 7.1 <settings>元素中的常见配置

设置参数	描述	有效值	默认值
cacheEnabled	该配置影响所有映射器中的缓存全局开关	true/false	false
lazyLoadingEnabled	延迟加载的全局开关。开启时，所有关联对象都会延迟加载。特定关联关系中可以通过设置 fetchType 属性来覆盖该项的开关状态	true/false	false
aggressiveLazyLoading	关联对象属性的延迟加载开关。当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之，每种属性都会按需加载	true/false	true
multipleResultSetsEnabled	是否允许单一语句返回多结果集（需要兼容驱动）	true/false	true
useColumnLabel	使用列标签代替列名。不同的驱动在这方面有不同的表现。具体可参考驱动文档或通过测试两种模式来观察所用驱动的行为	true/false	true

（续表）

设置参数	描述	有效值	默认值
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要驱动兼容。如果设置为 true，那么这个设置强制使用自动生成主键，尽管一些驱动不兼容，但仍可正常工作	true/false	false
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段或属性。NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集；FULL 会自动映射任意复杂的结果集（无论是否嵌套）	NONE、PARTIAL、FULL	PARTIAL
defaultExecutorType	配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）；BATCH 执行器将重用语句并执行批量更新	SIMPLE、REUSE、BATCH	SIMPLE
defaultStatementTimeout	设置超时时间，决定驱动等待数据库响应的秒数。没有设置值的时候，它取的是驱动默认的时间	任何正整数	没有设置
mapUnderscoreTocamelcase	是否开启自动驼峰命名规则（Camel-Case）映射	true/false	false
jdbcTypeForNull	当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。某些驱动需要指定列的 JDBC 类型，多数情况下直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER	NULL、VARCHAR、OTHER	OTHER

表 7.1 中介绍了<settings>元素中的常见配置，这些配置在配置文件中的使用方式如下。

```
<!--设置-->
<settings>
    <setting name="cacheEnabled" value="true" />
    <setting name="lazyLoadingEnabled" value="true" />
    <setting name="multipleResultsetsEnabled" value="true" />
    <setting name="useColumnLabel" value="true" />
    <setting name="useGeneratedKeys" value="false"/>
    <setting name="autoMappingBehavior" value="PARTIAL"/>
    ……
</settings>
```

上面所介绍的配置内容通常在需要时只配置少数几项即可。

7.2.3　<typeAliases>元素

<typeAliases>元素用于为配置文件中的 Java 类型设置一个简短的名字，即设置别名。别名的设置与 XML 配置相关，其使用的意义在于减少全限定类名的冗余。

使用<typeAliases>元素配置别名的方法如下：

```
<!--定义别名-->
<typeAliases>
```

```
    <typeAlias alias="user" type="com.ssm.po.User"/>
</typeAliases>
```

在上述示例中，<typeAliases>元素的子元素<typeAlias>中的 type 属性用于指定需要被定义别名的类的全限定名；alias 属性的属性值“user”就是自定义的别名，可以代替“com.ssm.po.User”使用在 MyBatis 文件的任何位置。如果省略 alias 属性，MyBatis 会默认将类名首字母小写后的名称作为别名。

当 POJO 类过多时，还可以通过自动扫描包的形式自定义别名，具体示例如下。

```
<!-使用自动扫描包来定义别名-->
<typeAliases>
    <package name="com.ssm.po"/>
</typeAliases>
```

在上述示例中，<typeAliases>元素的子元素<package>中的 name 属性用于指定要被定义别名的包，MyBatis 会将所有 com.ssm.po 包中的 POJO 类以首字母小写的非限定类名作为它的别名。

注 意

上述方式的别名只适用于没有使用注解的情况。如果在程序中使用了注解，那么别名为其注解的值，具体如下。

```
@Alias(value ="user")
    public class User {
        //user 的属性和方法
    ......
}
```

除了可以使用<typeAliases>元素自定义别名外，MyBatis 框架还默认为许多常见的 Java 类型（如数值、字符串、日期和集合等）提供相应的类型别名，如表 7.2 所示。

表 7.2 MyBatis 默认别名

别名	映射的类型	别名	映射的类型
_byte	byte	double	Double
_short	short	float	Float
_int	int	boolean	Boolean
_integer	int	date	Date
_long	long	decimal	BigDecimal
float	float	bigdecimal	BigDecimal
_double	double	object	Object
_boolean	boolean	map	Map
string	String	hashmap	HashMap
byte	Byte	list	List
long	Long	arraylist	ArrayList

（续表）

别名	映射的类型	别名	映射的类型
short	Short	collection	Collection
int	Integer	iterator	Iterator
integer	Integer		

提　示

表 7.2 所列举的别名可以在 MyBatis 中直接使用，但由于别名不区分大小写，因此在使用时要注意重复定义的覆盖问题。

7.2.4　<typeHandler>元素

MyBatis 在预处理语句（Prepared Statement）中设置一个参数或者从结果集（Resultset）中取出一个值时，都会用其框架内部注册了的 typeHandler（类型处理器）进行相关处理。typeHandler 的作用就是将预处理语句中传入的参数从 javaType（Java 类型）转换为 jdbcType（JDBC 类型），或者从数据库取出结果时将 jdbcType 转换为 javaType。

为了方便转换，MyBatis 框架提供了一些默认的类型处理器，如表 7.3 所示。

表 7.3　常用的类型处理器

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	java.Lang.Boolean,boolean	数据库兼容的 BOOLEAN
ByteTypeHandler	java.Jang.Byte,byte	数据库兼容的 NUMERIC 或 BYTE
ShortTypeHandler	java.lang.Short,short	数据库兼容的 NUMERIC 或 SHORT INTEGER
IntegerTypeHandler	java.lang.Integer,int	数据库兼容的 NUMERIC 或 INTEGER
LongTypeHandler	java.lang.Long,long	数据库兼容的 NUMERIC 或 LONG INTEGER
FloatTypeHandler	java.lang.Float,float	数据库兼容的 NUMERIC 或 FLOAT
DoubleTypeHandler	java.lang.Double,double	数据库兼容的 NUMERIC 或 DOUBLE
BigDecimalTypeHandler	java.math.BigDecimal	数据库兼容的 NUMERIC 或 DECIMAL
StringTypeHandler	java.langString	CHAR、VARCHAR
ClobTypeHandler	java.lang.String	CLOB、LONGVARCHAR
ByteArrayTypeHandler	byte[]	数据库兼容的字节流类型
BlobTypeHandler	byte[]	BLOB、LONGVARBINARY
DateTypeHandler	java.util.Date	TIMESTAMP
SqlTimestampTypeHandler	Java.sql.Timestamp	TIMESTAMP
SalDateTypeHandlel	java.sql.Date	DATE
sqlTimeTypeHandler	java.sql.Time	TIME

当 MyBatis 框架所提供的这些类型处理器不能够满足需求时，还可以通过自定义的方式对类型处理器进行扩展。自定义类型处理器可以通过实现 TypeHandler 接口或者继承 BaseTypeHandle 类来定义。<typeHandler>元素就是用于在配置文件中注册自定义的类型处理器的。它的使用方式有两种，

具体如下。

（1）注册一个类的类型处理器

```
<typeHandlers>
    <!--以单个类的形式配置-->
    <typeHandler handler="com.ssm.type.UsertypeHandlerl" />
</typeHandlers>
```

上述代码中，子元素<typeHandler>的 handler 属性用于指定在程序中自定义的类型处理器类。

（2）注册一个包中所有的类型处理器

```
<typeHandlers>
    <!-注册一个包中所有的 typeHandler，系统在启动时会自动扫描包下的所有文件-->
    <package name="com.ssm.type" />
</typeHandlers>
```

在上述代码中，子元素<package>的 name 属性用于指定类型处理器所在的包名，使用这种方式后，系统会在启动时自动扫描 com.ssm.type 包下所有的文件，并把它们作为类型处理器。

7.2.5 <objectFactory>元素

MyBatis 框架每次创建结果对象的新实例时，都会使用一个对象工厂（ObjectFactory）的实例来完成。MyBatis 中默认的 ObjectFactory 的作用就是实例化目标类，既可以通过默认构造方法实例化，也可以在参数映射存在的时候通过参数构造方法来实例化。

在通常情况下，我们使用默认的 ObjectFactory 即可。MyBatis 中默认的 ObjectFactory 是由 org.apache.ibatis.reflection.factory.DefaultObjectFactory 来提供服务的，大部分场景下都不用配置和修改。如果想覆盖 ObjectFactory 的默认行为，那么可以通过自定义 ObjectFactory 实现。

7.2.6 <plugins>元素

MyBatis 允许在已映射语句执行过程中的某一点进行拦截调用（通过插件来实现）。<plugins>元素的作用就是配置用户所开发的插件。关于插件的使用，本书不做详细讲解，有兴趣的读者请查找官方文档等资料自行学习。

7.2.7 <environments>元素

<environments>元素用于在配置文件中对环境进行配置。MyBatis 的环境配置实际上就是数据源的配置，可以通过<environments>元素配置多种数据源，即配置多种数据库。

【示例 7-2】使用<environments>元素进行环境配置的示例如下。

```
<environments default="development">
    <environment id="development">
        <!--使用 JDBC 事务管理-->
```

```
        <transactionManager type="JDBC"/>
        <!--配置数据源-->
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}" />
            <property name="url" value="${jdbc.url}" />
            <property name="username" value="${jdbc. username }" />
            <property name=" password " value="${jdbc. password }" />
        </datasource>
    </environment>
    ……
</environments>
```

在上述示例代码中，<environments>元素是环境配置的根元素，它包含一个 default 属性，该属性用于指定默认的环境 ID。<environment>是<environments>元素的子元素，它可以被定义多个，其 id 属性用于表示所定义环境的 ID 值。在<environment>元素内，包含事务管理和数据源的配置信息，其中<transactionManager>元素用于配置事务管理，它的 type 属性用于指定事务管理的方式，即使用哪种事务管理器；<dataSource>元素用于配置数据源，它的 type 属性用于指定使用哪种数据源。

在 MyBatis 中，可以配置两种类型的事务管理器，分别是 JDBC 和 MANAGED。

- JDBC：此配置直接使用 JDBC 的提交和回滚设置，依赖从数据源得到的连接来管理事务的作用域。
- MANAGED：此配置从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期。在默认情况下，它会关闭连接，但一些容器并不希望这样，为此可以将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。

注　意
如果项目中使用 Spring+MyBatis，就没有必要在 MyBatis 中配置事务管理器，因为实际开发中会使用 Spring 自带的管理器来实现事务管理。

对于数据源的配置，MyBatis 框架提供了 UNPOOLED、POOLED 和 JNDI 三种数据源类型。

- UNPOOLED：配置此数据源类型后，在每次被请求时会打开和关闭连接。它对没有性能要求的简单应用程序是一个很好的选择。
- POOLED：此数据源利用“池”的概念将 JDBC 连接对象组织起来，避免在创建新的连接实例时需要初始化和认证的时间。这种方式使得并发 Web 应用可以快速地响应请求，是当前流行的处理方式。
- JNDI：此数据源可以在 EJB 或应用服务器等容器中使用。容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。

7.2.8　<mappers>元素

在配置文件中，<mappers>元素用于指定 MyBatis 映射文件的位置，一般可以使用以下 4 种方法

引入映射器文件，具体如下所示。

（1）使用类路径引入

```
<mappers>
    <mapper resource="com/ssm/mapper/UserMapper.xmI"/>
</mappers>
```

（2）使用本地文件路径引入

```
<mappers>
    <mapper url=file: ///D:/com/ssm/mapper/UserMapper.xml"/>
</mappers>
```

（3）使用接口类引入

```
<mappers>
    <mapper class="com.ssm.mapper.UserMapper"/>
</mappers>
```

（4）使用包名引入

```
<mappers>
    <package name=""com.ssm.mapper"/>
</mappers>
```

7.3 映射文件

映射文件是 MyBatis 框架中十分重要的文件。在映射文件中，<mapper>元素是映射文件的根元素，其他元素都是它的子元素。映射文件中的主要元素如下所示。

```
<mapper>
   <!-- 映射查询语句，可自定义参数、返回结果集等 -->
   <select/>
   <!-- 映射插入语句，执行后返回一个整数，代表插入的条数 -->
   <insert/>
   <!-- 映射更新语句，执行后返回一个整数，代表更新的条数 -->
   <update/>
   <!-- 映射删除语句，执行后返回一个整数，代表删除的条数 -->
   <delete/>
   <!-- 用于定义一部分 SQL，然后可被其他语句引用此 SQL -->
   <sql/>
   <!-- 给定命名空间的缓存配置 -->
   <cache/>
   <!-- 其他命名空间缓存配置的引用 -->
   <cache-ref/>
   <!-- 用于描述如何从数据库结果集中加载对象 -->
   <resultMap/>
```

```
</mapper>
```

7.3.1　<select>元素

<select>元素用于映射查询语句，从数据库中读取数据，并组装数据给业务开发人员。示例如下：

```
<select id="findUserById" parameterType="Integer" resultType="com.ssm.po.User">
   select * from t_user where id=#{id}
</select>
```

上述语句中的唯一标识为 findUserById，它接收一个 Integer 类型的参数，并返回一个 User 类型的对象。

在<select>元素中，除了上述示例代码中的几个属性外，还有其他可以配置的属性，如表 7.4 所示。

表 7.4　<select>元素的常用属性

属性	说明
id	表示命名空间中的唯一标识符，常与命名空间组合起来使用。组合后如果不唯一，MyBatis 就会抛出异常
parameterType	该属性表示传入 SQL 语句的参数类的全限定名或者别名。它是一个可选属性，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数。其默认值是 unset（依赖于驱动）
resultType	从 SQL 语句中返回的类型的类的全限定名或者别名。如果是集合类型，那么返回的应该是集合可以包含的类型，而不是集合本身。返回时可以使用 resultType 或 resultMap 之一
resultMap	表示外部 resultMap 的命名引用。返回时可以使用 resultType 或 resultMap 之一
flushCache	表示在调用 SQL 语句之后，是否需要 MyBatis 清空之前查询的本地缓存和二级缓存。其值为布尔类型（true/false），默认值为 false。如果设置为 true，那么任何时候只要 SQL 语句被调用，都会清空本地缓存和二级缓存
useCache	用于控制二级缓存的开启和关闭。其值为布尔类型（true/false），默认值为 true，表示将查询结果存入二级缓存中
timeout	用于设置超时参数，单位为秒。超时时将抛出异常
fetchSize	获取记录的总条数设定，其默认值是 unset（依赖于驱动）
statementType	用于设置 MyBatis 使用哪个 JDBC 的 Statement 工作，其值为 STATEMENT、PREPARED（默认值）或 CALLABLE，分别对应 JDBC 中的 Statement、PreparedStatement 和 CallableStatement
resultSetType	表示结果集的类型，其值可设置为 FORWARD_ONLY、SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE，默认值是 unset（依赖于驱动）

7.3.2 <insert>元素

<insert>元素用于映射插入语句，在执行完元素中定义的 SQL 语句后，会返回一个表示插入记录数的整数。<insert>元素的配置示例如下：

```
<insert id="addUser" parameterType="com.ssm.po.User" flushCache="true"
    statementType="PREPARED"  keyProperty="id" keyColumn=""
    useGeneratedKeys="" timeout="20">
      insert into    t_user(username,jobs,phone)values(#{username},#{jobs},#{phone})
</insert>
```

从上述示例代码中可以看出，<insert>元素的属性与<select>元素的属性大部分相同，但还包含 3 个特有属性（仅对 insert 和 update 有用），如表 7.5 所示。

表 7.5 < insert>元素的常用属性

属性	说明
keyProperty	此属性的作用是将插入或更新操作时的返回值赋给 PO 类的某个属性，通常会设置为主键对应的属性。如果需要设置联合主键，可以在多个值之间用逗号隔开
keyColumn	此属性用于设置第几列是主键，当主键列不是表中的第一列时需要设置。在需要主键联合时，值可以用逗号隔开
useGeneratedKeys	此属性会使 MyBatis 用 JDBC 的 getGeneratedKeys()方法来获取由数据库内部产生的主键，如 MSQL 和 SQL Server 等自动递增的字段，其默认值为 false

执行插入操作后，很多时候我们会需要返回插入成功的数据生成的主键值，此时就可以通过上面所讲解的 3 个属性来实现。

【示例 7-3】如果使用的数据库支持主键自动增长（如 MSQL），那么可以通过 keyProperty 属性指定 PO 类的某个属性接收主键返回值（通常会设置到 id 属性上），然后将 useGeneratedKeys 的属性值设置为 true。使用上述配置执行插入后，会返回插入成功的行数以及插入行的主键值。可以通过如下代码测试。

```
@Test
public void addUserTest() throws Exception {
   String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory =
                  new SqlSessionFactoryBuilder().build(inputStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    User user = new User();
    user.setUsername("jack");
    user.setJobs("worker");
    user.setPhone("13324585254");
    int rows = sqlSession.insert("com.ssm.mapper.UserMapper.addUser", user);
   //输出插入数据的主键 id 值
   System.out.println(user.getId());
    if (rows > 0) {
```

```
        System.out.println("成功添加" + rows + "条数据！");
    } else {
        System.out.println("添加数据失败！");
    }
    sqlSession.commit();
    sqlSession.close();
}
```

如果使用的数据库不支持主键自动增长（如 Oracle），或者支持增长的数据库取消了主键自增的规则，就可以使用 MyBatis 提供的另一种方式来自定义生成主键，具体配置示例如下。

```
<insert id="insertUser" parameterType="com.ssm.po.User">
   <selectKey keyProperty="id" resultType="Integer" order="BEFORE">
   select if(max(id) is null, 1, max(id)+1) as newId from t_user
   </select>
   insert into t_user(id,username,jobs,phone) values(#{id},#{username},#{jobs},#{phone})
</insert>
```

在执行上述示例代码时，<selectKey>元素会首先运行，它会通过自定义的语句来设置数据表中的主键（如果 t_uesr 表中没有记录，就将 id 设置为 1，否则将 id 的最大值加 1 作为新的主键），然后调用插入语句。

<selectKey>元素在使用时可以设置以下几种属性。

```
<selectKey
   keyProperty="id"
   resultType="Integer"
   order="BEFORE"
   statement="PREPARED">
```

在上述<selectKey>元素的几个属性中，keyProperty、resultType 和 statement 的作用与前面讲解的相同。order 属性可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会先执行<selectKey>元素中的配置来设置主键，再执行插入语句；如果设置为 AFTER，那么它会先执行插入语句，再执行<selectKey>元素中的配置内容。

7.3.3　<update>元素和<delete>元素

<update>元素和<delete>元素的使用比较简单，它们的属性配置也基本相同（<delete>元素中不包含表 7.5 中的 3 个属性），其常用属性如下所示。

```
<update
     id="updateUser"
     parameterType="com.ssm.po.User"
     flushCache="true"
     statementType="PREPARED"
     timeout="20">
<delete
     id="deleteUser"
```

```
    parameterType="com.ssm.po.User"
    flushCache="true"
    statementType="PREPARED"
    timeout="20">
```

从上述配置代码中可以看出，<update>元素和<delete>元素的属性基本与<select>元素中的属性一致。与<insert>元素一样，<update>元素和<delete>元素在执行完之后，也会返回一个表示影响记录条数的整数，其使用示例如下。

```
<!--更新用户信息 -->
<update id="updateUser" parameterType="com.ssm.po.User">
    update t_user set username=#{username},jobs=#{jobs},phone=#{phone} where id=#{id}
</update>
<!--删除用户信息-->
<delete id="deleteUser" parameterType="Integer">
    delete from t_user where id=#{id}
</delete>
```

7.3.4 <sql>元素

在一个映射文件中，通常需要定义多条 SQL 语句，这些 SQL 语句的组成可能有一部分是相同的（如多条 select 语句中都查询相同的 id、username、jobs 字段），如果每一个 SQL 语句都重写一遍相同的部分，势必会增加代码量，导致映射文件过于臃肿。那么有没有什么办法将这些 SQL 语句中相同的组成部分抽取出来，然后在需要的地方引用呢？答案是肯定的，我们可以在映射文件中使用 MyBatis 提供的<sql>元素来解决上述问题。

<sql>元素的作用是定义可重用的 SQL 代码片段，然后在其他语句中引用这一代码片段。

例如，定义一个包含 id、username、jobs 和 phone 字段的代码片段：

```
<sql id="user Columns">id,username,jobs, phone</sql>
```

这一代码片段可以包含在其他语句中使用，具体如下：

```
<select id="findUserById" parameterType="Integer" resultType="com.ssm.po.User">
    select <include refid="user Columns">
    from t_user
    where id=#{id}
</select>
```

在上述代码中，使用<include>元素的 refid 属性引用了自定义的代码片段，refid 属性值为自定义代码片段的 id。

上面的示例只是一个简单的引用查询。在实际开发中，可以更加灵活地定义 SQL 片段，读者可以查找相关资料进一步了解。

7.3.5　<resultMap>元素

<resultMap>元素表示结果映射集，主要作用是定义映射规则、级联更新以及定义类型转化器等。< resultMap>元素中包含一些子元素，元素结构如下所示。

```
<!-- resultMap 的元素结构-->
<resultMap type="" id="">
    <constructor>  <!--类在实例化时，用来注入结果到构造方法中-->
        <idArg />    <!--ID 参数，标记结果作为 ID-->
        <arg />      <!--注入到构造方法的一个普通结果-->
    </constructor>
    <id />            <!--用于表示哪个列是主键-->
    <result />        <!--注入到字段或 Javabean 属性的普通结果-->
    <association property="" />    <!--用于一对一关联-->
    <collection property="" />      <!--用于一对多关联-->
    <discriminator javaType="">   <!--使用结果值来决定使用哪个结果映射-->
         <case value="" />      <!--基于某些值的结果映射-->
    </discriminator>
</resultMap>
```

<resultMap>元素的 type 属性表示需要映射的 POJO，id 属性是这个 resultMap 的唯一标识。它的子元素<constructor>用于配置构造方法（当一个 POJO 中未定义无参的构造方法时，就可以使用<constructor>元素进行配置）。子元素<id>用于表示哪个列是主键，而<result>用于表示 POJO 和数据表中普通列的映射关系。<association>和<collection>用于处理多表时的关联关系，而<discriminator>元素主要用于处理一个单独的数据库查询返回很多不同数据类型结果集的情况。

在默认情况下，MyBatis 程序在运行时会自动地将查询到的数据与需要返回的对象的属性进行匹配赋值（需要表中的列名与对象的属性名称完全一致）。然而实际开发时，数据表中的列和需要返回的对象的属性可能不会完全一致，这种情况下 MyBatis 是不会自动赋值的。此时，就可以使用<resultMap>元素进行处理，示例代码 UserMapper.xml 如下所示。

```
<?xml version="1. 0" encoding="UTE-8"?>
<!-- DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
    <mapper name space="com.ssm.mapper.UserMapper">
         <resultMap type="com.ssn.po.User" id="resultMap">
            <id property="id" column="t_id"/>
            <result property="name" column="t_username"/>
            <result property="age" column="t_age"/>
    </resultMap>
    <select id="findAllUser" resultMap="resultMap">
         select * from t_user
    </select>
```

在上述代码中，<resultMap>的子元素<id>和<result>的 property 属性表示 User 类的属性名，

column 属性表示数据表 t_user 的列名。<select>元素的 resultMap 属性表示引用上面定义的 resultMap。接下来可以在配置文件 mybatis-config.xml 中引入 UserMapper.xml。

除此之外，还可以通过<resultMap>元素中的<association>和<collection>处理多表时的关联关系，这将在后续章节中进行详细讲解。

7.4 习 题

1. 请简述 MyBatis 核心对象 SqlSessionFactory 的获取方式。
2. 请简述 MyBatis 映射文件的主要元素及其作用。

第 8 章

动态 SQL

MyBatis 提供对 SQL 语句动态组装的功能，能很好地解决开发人员在使用 JDBC 或框架进行数据库开发时手动拼装 SQL 这一非常麻烦且痛苦的工作。动态 SQL 是 MyBatis 的强大特性之一，MyBatis 3 采用了功能强大的基于 OGNL 的表达式来完成动态 SQL。

本章主要涉及的知识点如下：

- <if>：判断语句，用于单条件分支判断。
- <choose>(<when>、<otherwise>)：用于多条件分支判断。
- <where>、<trim>、<set>：辅助元素，用于处理一些 SQL 拼装、特殊字符问题。
- <foreach>：循环语句，常用于 in 语句等列举条件中。
- <bind>：从 OGNL 表达式中创建一个变量，并将其绑定到上下文，常用于模糊查询的 SQL 中。

8.1 <if>元素

在 MyBatis 中，<if>元素是常用的判断语句，主要用于实现某些简单的条件选择。在实际应用中，我们可能会通过多个条件来精确地查询某个数据。例如，要查找某个用户信息，可以通过姓名和职业来查找用户，也可以不填写职业，直接通过姓名来查找用户，还可以都不填写而查询出所有用户，此时姓名和职业就是非必需条件。类似于这种情况，在 MyBatis 中就可以通过<if>元素来实现。

【示例 8-1】下面通过一个具体的案例来演示<if>元素的使用。

（1）在 Eclipse 中创建一个名为 chapter08 的 Web 项目。

（2）将第 6 章中 chapter06 项目中的 JAR 包和 src 目录下的文件复制到 chapter08 中。

（3）将配置文件中的数据库信息修改为外部引用的形式，即在项目的 src 目录下创建一个名称为 db.properties 的配置文件，编辑代码，如下所示。

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql: //localhost: 3306/db_mybatis
jdbc.username=root
jdbc.password=root
```

然后在 MyBatis 配置文件 mybatis-config.xml 中配置<properties/>属性，并修改配置文件中数据库连接的信息，修改完成后 mybatis-config.xml 配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
 <configuration>
 <properties resource="db.properties" />
 <environments default="mysql">
     <environment id="mysql">
         <transactionManager type="JDBC" />
         <dataSource type="POOLED">
             <!--数据库驱动 -->
             <property name="driver" value="${jdbc.driver}" />
             <!--连接数据库的 url -->
             <property name="url" value="${jdbc.url}" />
             <!--连接数据库的用户名 -->
             <property name="username" value="${jdbc.username}" />
             <!--连接数据库的密码-->
             <property name="password" value="${jdbc.password}" />
         </dataSource>
     </environment>
 </environments>
 <mappers>
     <mapper resource="com/ssm/mapper/UserMapper.xml" />
 </mappers>
</configuration>
```

（4）创建一个 com.ssm.util 包，在该包下创建工具类 MybatisUtil，在其中定义获取 SqlSession 的方法 getSession()，如文件 8.1 所示。

文件 8.1　MybatisUtil.java

```
package com.ssm.util;
import java.io.IOException;
import java.io.Reader;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
```

```
public class MybatisUtils{
    private static SqlSessionFactory sqlSessionFactory=null;
    static{
        try {
        String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            sqlSessionFactory=new SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
 // 获取 SqlSession 的方法
    public static SqlSession getSession(){
        return sqlSessionFactory.openSession();
    }
}
```

搭建后的项目文件结构如图 8.1 所示。

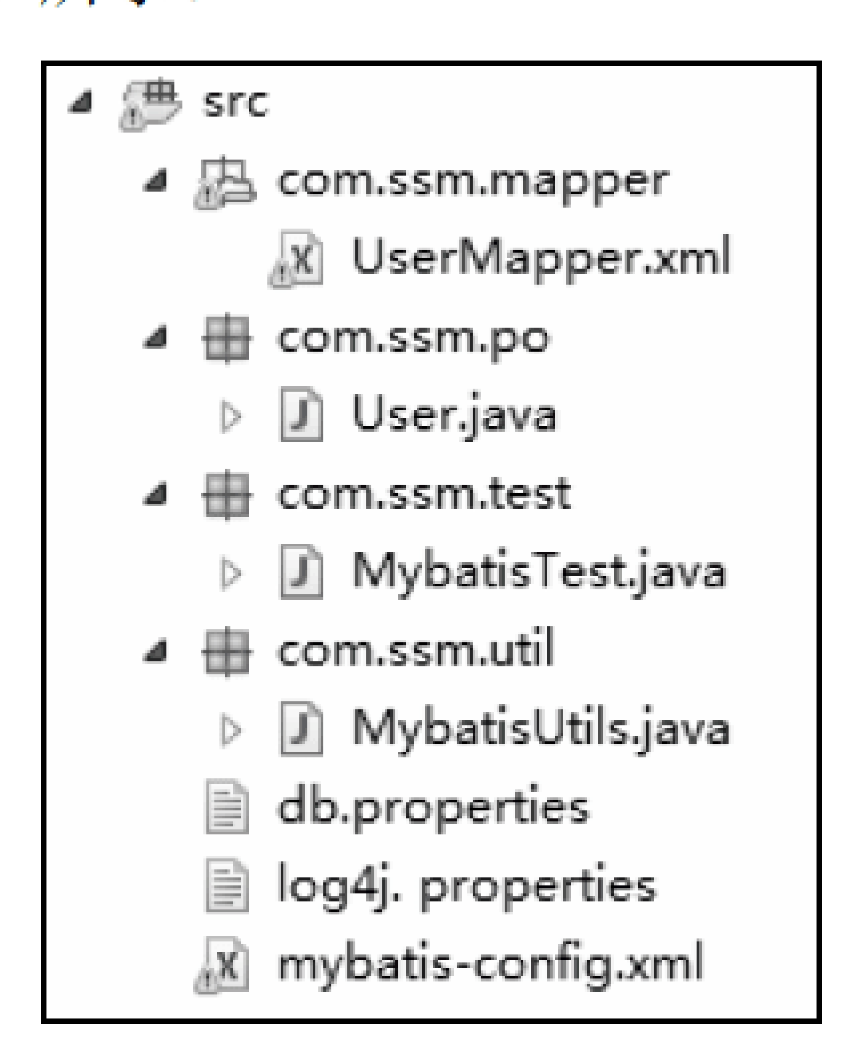


图 8.1　chapter08 项目文件结构

（5）修改映射文件 UserMapper.xml，在映射文件中使用<if>元素编写根据用户姓名（username）和职业（jobs）组合条件查询用户信息列表的动态 SQL，如文件 8.2 所示。

文件 8.2　UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.UserMapper">
    <!--<if>元素使用 -->
     <select id="findUserByNameAndJobs" parameterType="com.ssm.po.User"
     resultType="com.ssm.po.User">
         select * from t_user where 1=1
     <if test="username !=null and username !=''">
         and username like concat('%',#{username},'%')
     </if>
```

```
        <if test="jobs !=null and jobs !=''">
            and jobs=#{jobs}
        </if>
    </select>
</mapper>
```

在文件 8.2 中，使用<if>元素的 test 属性分别对 username 和 jobs 进行了非空判断（test 属性多用于条件判断语句中，用于判断真假，大部分的场景中都是进行非空判断的，有时也需要判断字符串、数字和枚举等），如果传入的查询条件非空，就进行动态 SQL 组装。

（6）在测试类 MybatisTest 中，编写测试方法 findUserByNameAndJobsTest()，如文件 8.3 所示。

文件 8.3　MybatisTest.java

```
package com.ssm.test;
import java.io.InputStream;
import java.util.List;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Test;
import com.ssm.po.User;
import com.ssm.util.MybatisUtils;
public class MybatisTest {
/*
     * 根据用户姓名和职业组合条件查询用户信息列表
     */
    @Test
    public void findUserByNameAndJobsTest() throws Exception {
     //通过工具类生成 SqlSession 对象
     SqlSession sqlSession = MybatisUtils.getSession();
     //创建 User 对象，封装需要组合查询的条件
     User user=new User();
     user.setUsername("zhangsan");
     user.setJobs("teacher");
     //执行 SqlSession 的查询方法，返回结果集
     List<User> users =
      sqlSession.selectList("com.ssm.mapper.UserMapper.findUserByNameAndJobs", user);
     //输出查询结果
     for (User u : users) {
         System.out.println(u.toString());
     }
     sqlSession.close();
 }
}
```

在文件 8.3 的 findUserByNameAndJobsTest()方法中，首先通过 MybatisUtils 工具类获取了 SqlSession 对象，然后使用 User 对象封装了用户名为 zhangsan 且职业为 teacher 的查询条件，并通过 SqlSession 对象的 selectList()方法执行多条件组合的查询操作。最后，程序执行完毕时关闭了 SqlSession 对象。执行 findUserByNameAndJobsTest()方法后，控制台的输出结果如图 8.2 所示。

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Problems  Console  Progress  JUnit
<terminated> MybatisTest.findUserByNameAndJobsTest [JUnit] C:\Program Files\Java\jdk1.7.0_72\bin\javaw.exe (2018年11月2日 上午11:45:43)
User [id=1, username=zhangsan, jobs=teacher, phone=13907998372]
```

图 8.2　运行结果

从图 8.2 可以看出，已经查询出了 username 为 zhangsan 且 jobs 为 teacher 的用户信息。先将封装到 User 对象中的 zhangsan 和 teacher 两行代码注释掉，然后再次执行 findUserByNameAndJobsTest()方法，控制台的输出结果如图 8.3 所示。

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Problems  Console  Progress  JUnit
<terminated> MybatisTest.findUserByNameAndJobsTest [JUnit] C:\Program Files\Java\jdk1.7.0_72\bin\javaw.exe (2018年11月2日 下午2:02:26)
User [id=1, username=zhangsan, jobs=teacher, phone=13907998372]
User [id=2, username=lisi, jobs=worker, phone=13907396542]
User [id=3, username=wangwu, jobs=doctor, phone=13817348729]
```

图 8.3　运行结果

从图 8.3 可以看到，当未传递任何参数时，会将数据表中的所有数据查出来。这就是<if>元素的使用。

8.2　<choose>、<when>和<otherwise>元素

在使用<if>元素时，只要 test 属性中的表达式为 true，就会执行元素中的条件语句，但是在实际应用中，有时只需要从多个选项中选择一个执行。例如，若用户姓名不为空，则只根据用户姓名进行筛选；若用户姓名为空，而用户职业不为空，则只根据用户职业进行筛选；若用户姓名和用户职业都为空，则要求查询出所有电话不为空的用户信息。

此种情况下，使用<if>元素进行处理是非常不合适的，可以使用<choose>、<when>、<otherwise>元素进行处理，类似于在 Java 语言中使用 switch...case…default 语句。

【示例 8-2】使用<choose>、<when>、<otherwise>元素组合实现上面的情况。

（1）在映射文件 UserMapper.xml 中，使用<choose>、<when>、<otherwise>元素执行上述情况的动态 SQL 代码如下所示。

```
<!--<choose>(<when>、<otherwise>)元素使用 -->
    <select id="findUserByNameOrJobs" parameterType="com.ssm.po.User"
        resultType="com.ssm.po.User">
         select * from t_user where 1=1
     <choose>
         <when test="username !=null and username !=''">
             and username like concat('%',#{username},'%')
```

```
            </when>
            <when test="jobs !=null and jobs !=''">
                and jobs=#{jobs}
            </when>
            <otherwise>
                and phone is not null
            </otherwise>
        </choose>
    </select>
```

在上述代码中，使用了<choose>元素进行 SQL 拼接，若第一个<when>元素中的条件为真，则只动态组装第一个<when>元素内的 SQL 片段；否则继续向下判断第二个<when>元素中的条件是否为真，以此类推；若前面所有 when 元素中的条件都不为真，则只组装<otherwise>元素内的 SQL 片段。

（2）在测试类 MybatisTest 中，编写测试方法 findUserByNameOrJobsTest()，其代码如下所示。

```
/*
     * 根据用户姓名或者职业组合条件查询用户信息列表
     */
    @Test
    public void findUserByNameOrJobsTest() throws Exception {
     SqlSession sqlSession = MybatisUtils.getSession();
     User user=new User();
     user.setUsername("zhangsan");
     user.setJobs("teacher");
     //执行 SqlSession 的查询方法，返回结果集
     List<User> users =
      sqlSession.selectList("com.ssm.mapper.UserMapper.findUserByNameOrJobs", user);
     for (User u : users) {
         System.out.println(u.toString());
     }
     sqlSession.close();
    }
```

执行上述方法后，结果与图 8.2 相同，不过虽然同时传入了姓名和职业两个查询条件，但 MyBatis 所生成的 SQL 是动态组装用户姓名进行条件查询的。如果将上述代码中的“user.setUsername("zhangsan");”删除或者注释掉，然后再次执行，这时 MyBatis 生成的 SQL 组装用户职业进行条件查询，同样会查询出用户信息。如果将设置客户姓名和职业参数值的两行代码都注释掉，那么程序的执行结果如图 8.3 所示，MyBatis 的 SQL 组装<otherwise>元素中的 SQL 片段进行条件查询。

8.3　<where>、<trim>元素

在前两节的案例中，映射文件中编写的 SQL 后面都加入了“where1=1”的条件，是为了保证当条件不成立时拼接起来的 SQL 语句在执行时不会报错，即使得 SQL 不出现语法错误。那么在 MyBatis 中，有没有什么办法不用加入“1=1”这样的条件，也能使拼接后的 SQL 成立呢？针对这种情况，MyBatis 提供了<where>元素。

【示例 8-3】以 8.1.1 小节的案例为例，将映射文件中的“where 1=1”条件删除，使用<where>元素替换后的代码如下所示。

```
<!--<if>、<where>元素使用 -->
    <select id="findUserByNameAndJobs" parameterType="com.ssm.po.User"
    resultType="com.ssm.po.User">
    select * from t_user
      <where>
        <if test="username !=null and username !=''">
            and username like concat('%',#{username},'%')
        </if>
        <if test="jobs !=null and jobs !=''">
            and jobs=#{jobs}
        </if>
     </where>
    </select>
```

上述配置代码中，使用<where>元素对“where 1=1”条件进行了替换，<where>元素会自动判断组合条件下拼装的 SQL 语句，只有<where>元素内的条件成立时，才会在拼接 SQL 中加入 where 关键字，否则将不会添加；即使 where 之后的内容有多余的“AND”或“OR”，<where>元素也会自动将它们去除。除了使用<where>元素外，还可以通过<trim>元素来定制需要的功能，上述代码可以修改为如下形式。

```
<!--<if>、<trim>元素的使用 -->
    <select id="findUserByNameAndJobs" parameterType="com.ssm.po.User"
    resultType="com.ssm.po.User">
    select * from t_user
      <trim prefix="where" prefixOverrides="and">
        <if test="username !=null and username !=''">
            and username like concat('%',#{username},'%')
        </if>
        <if test="jobs !=null and jobs !=''">
            and jobs=#{jobs}
        </if>
     </trim>
    </select>
```

上述配置代码中，同样使用<trim>元素对“whee 1=1”条件进行了替换，<trim>元素的作用是

去除一些特殊的字符串，它的 prefix 属性代表的是语句的前缀（这里使用 where 来连接后面的 SQL 片段），而 prefixOverrides 属性代表的是需要去除的那些特殊字符串（这里定义了要去除 SQL 中的 and），上面的写法和使用<where>元素基本是等效的。

8.4 <set>元素

在 Hibernate 中，如果想要更新某一个对象，就需要发送所有的字段给持久化对象，然而实际应用中会存在只需要更新某一个或几个字段。为了让程序只更新需要更新的字段，MyBatis 提供了<set>元素来完成这一工作。<set>元素主要用于更新操作，主要作用是在动态包含的 SQL 语句前输出一个 SET 关键字，并将 SQL 语句中最后一个多余的逗号去除。

【示例 8-4】以更新操作为例，使用<set>元素对映射文件中更新用户信息的 SQL 语句进行修改的代码如下所示。

```
<!-- <set>元素 -->
    <update id="updateUser" parameterType="com.ssm.po.User">
     update t_user
     <set>
         <if test="username !=null and username !=''">
             username=#{username}
         </if>
         <if test="jobs !=null and jobs !=''">
             jobs=#{jobs}
         </if>
         <if test="phone !=null and phone !=''">
             phone=#{phone}
         </if>
     </set>
     where id=#{id}
    </update>
```

在上述配置的 SQL 语句中，使用了<set>和<if>元素相结合的方式来组装 update 语句。其中<set>元素会动态前置 SET 关键字，同时消除 SQL 语句中最后一个多余的逗号；<if>元素用于判断相应的字段是否传入值，如果传入的更新字段非空，就将此字段进行动态 SQL 组装，并更新此字段，否则此字段不执行更新。

注 意

在映射文件中使用<set>和<if>元素组合进行 update 语句动态 SQL 组装时，如果<set>元素内包含的内容都为空，就会出现 SQL 语法错误。所以在使用<set>元素进行字段信息更新时，要确保传入的更新字段不能都为空。

8.5 <foreach>元素

MyBatis 中已经提供了一种用于数组和集合循环遍历的方式，那就是使用<foreach>元素。假设在一个用户表中有 1000 条数据，现在需要将 id 值小于 100 的用户信息全部查询出来，就可以通过<foreach>元素来解决。

<foreach>元素通常在构建 IN 条件语句时使用，其使用方式如下。

```
<!--<foreach>元素的使用 -->
    <select id="findUserByIds" parameterType="List" resultType="com.ssm.po.User">
        select * from t_user where id in
     <foreach item="id" index="index" collection="list" open="(" separator="," close=")">
       #{id}
     </foreach>
    </select>
```

在上述代码中，使用<foreach>元素对传入的集合进行遍历以及动态 SQL 组装。关于<foreach>元素中使用的几种属性的描述具体如下：

- item：配置的是循环中当前的元素。
- index：配置的是当前元素在集合中的位置下标。
- collection：配置的 list 是传递过来的参数类型（首字母小写），可以是一个 array、list（或 collection）、Map 集合的键、POJO 包装类中的数组或集合类型的属性名等。
- open 和 close：配置的是以什么符号将这些集合元素包装起来。
- separator：配置的是各个元素的间隔符。

注　意

可以将任何可迭代对象（如列表、集合等）和任何字典或者数组对象传递给<foreach>作为集合参数。当使用可迭代对象或者数组时，index 是当前迭代的次数，item 的值是本次迭代获取的元素。当使用字典（或者 MapEntry 对象的集合）时，index 是键，item 是值。

【示例 8-5】对<foreach>元素的使用进行测试。在测试类 MybatisTest 中，编写测试方法 findUserByIdsTest()，其代码如下所示。

```
/*
     * 根据用户编号批量查询用户信息
     */
    @Test
    public void findUserByIdsTest(){
     SqlSession sqlSession = MybatisUtils.getSession();
     //创建 List 集合，封闭查询 id
     List<Integer> ids=new ArrayList<Integer>();
     ids.add(1);
     ids.add(2);
```

```
    //执行 SqlSession 的查询方法，返回结果集
     List<User> users =
              sqlSession.selectList("com.ssm.mapper.UserMapper.findUserByIds", ids);
     for (User user : users) {
         System.out.println(user.toString());
     }
     sqlSession.close();
    }
```

在上述代码中，执行查询操作时传入了一个客户编号集合 ids。执行 findUserByIdsTest()方法后，控制台的输出结果如图 8.4 所示。从中可以看出，成功批量地查询出对应的用户信息。

```
<terminated> MybatisTest.findCustomerByIdsTest [JUnit] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年11月2日 下午9:18:52)
User [id=1, username=zhangsan, jobs=teacher, phone=13907998372]
User [id=2, username=lisi, jobs=worker, phone=13907396542]
```

图 8.4　运行结果

在使用<foreach>时，最关键、最容易出错的就是 collection 属性，该属性是必须指定的，而且在不同情况下该属性的值是不一样的，主要有以下 3 种情况。

- 如果传入的是单参数且参数类型是一个数组或者 List 的时候，collection 属性值分别为 array、list（或 collection）。
- 如果传入的参数有多个，就需要把它们封装成一个 Map，当然单参数也可以封装成 Map 集合，这时 collection 属性值就为 Map 的键。
- 如果传入的参数是 POJO 包装类，collection 属性值就为该包装类中需要进行遍历的数组或集合的属性名。

在设置 collection 属性值的时候，必须按照实际情况配置，否则程序就会出现异常。

8.6 <bind>元素

在进行模糊查询编写 SQL 语句的时候，若使用“${}”进行字符串拼接，则无法防止 SQL 注入问题；若使用 concat 函数进行拼接，则只针对 MySQL 数据库有效；若使用的是 Oracle 数据库，则要使用连接符号“||”。这样，映射文件中的 SQL 就要根据不同的情况提供不同形式的实现，显然是比较麻烦的，且不利于项目的移植。为此，MyBatis 提供了<bind>元素来解决这一问题。我们完全不必使用数据库语言，只要使用 MyBatis 的语言即可与所需参数连接。

MyBatis 的<bind>元素可以通过 OGNL 表达式来创建一个上下文变量，其使用方式如下所示。

```
<!--<bind>元素的使用：根据用户姓名模糊查询用户信息 -->
    <select id="findUserByName2" parameterType="com.ssm.po.User"
       resultType="com.ssm.po.User">
     <!--_parameter.getUsername()也可以直接写成传入的字段属性名，即 username  -->
     <bind name="p_username" value="'%'+_parameter.getUsername()+'%'"/>
```

```
    select * from t_user
    where username like #{p_username}
   </select>
```

上述配置代码中，使用<bind>元素定义了一个 name 为 p_username 的变量，<bind>元素中 value 的属性值就是拼接的查询字符串，其中_parameter.getUsername()表示传递进来的参数（也可以直接写成对应的参数变量名，如 username）。在 SQL 语句中，直接引用<bind>元素的 name 属性值即可进行动态 SQL 组装。

【示例 8-6】在测试类 MybatisTest 中，编写测试方法 findUserByName2()进行测试，其代码如下所示。

```
/*
    * 根据用户姓名模糊查询用户信息
    */
   @Test
   public void findUserByName2(){
    SqlSession sqlSession = MybatisUtils.getSession();
    User user=new User();
    user.setUsername("s");
    List<User> users =
          sqlSession.selectList("com.ssm.mapper.UserMapper.findUserByName2", user);
    for (User u : users) {
         System.out.println(u.toString());
    }
    sqlSession.close();
   }
```

执行后，在控制台的输出结果如图 8.5 所示。从中可以看出，使用 MyBatis 的<bind>元素已经完成了动态 SQL 组装，并成功模糊查询出了用户信息。

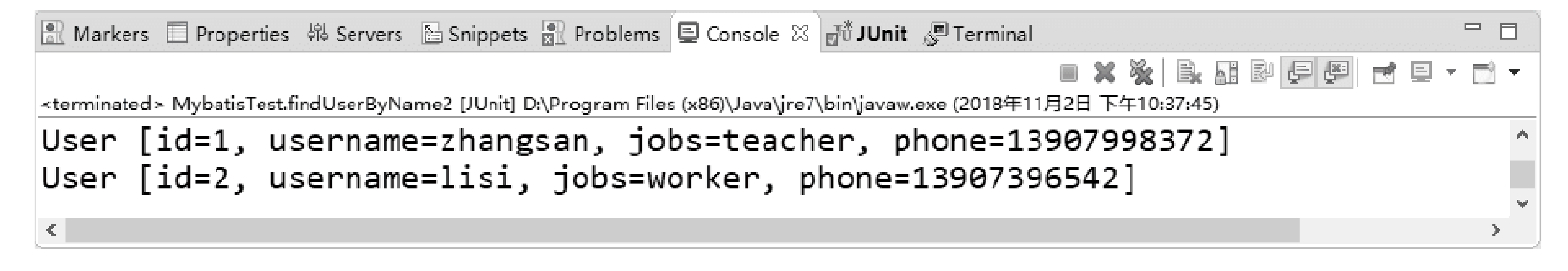


图 8.5　运行结果

8.7　习　　题

1. 请简述 MyBatis 框架动态 SQL 中的主要元素及其作用。
2. 请简述 MyBatis 框架动态 SQL 中<foreach>元素 collection 属性的注意事项。

第 9 章

MyBatis 的关联映射

在实际应用中，对数据库的操作会涉及多张表，这在面向对象中就涉及对象与对象之间的关联关系。针对多表之间的操作，MyBatis 提供了关联映射，通过关联映射来处理对象与对象之间的关联关系。

本章主要涉及的知识点如下：

- 关联关系概述：数据表之间以及对象之间的 3 种关联关系。
- 关联关系：一对一、一对多和多对多关联映射的使用、关联关系中的嵌套查询和嵌套结果。

9.1 关联关系概述

在关系型数据库中，多表之间存在 3 种关联关系，分别为一对一、一对多和多对多。

- 一对一：在任意一方引入对方主键作为外键。
- 一对多：在“多”的一方添加“一”的一方的主键作为外键。
- 多对多：产生中间关系表，引入两张表的主键作为外键，两个主键成为联合主键或使用新的字段作为主键。

对象之间也存在 3 种关联关系。

- 一对一的关系：在本类中定义对方类型的对象，比如 A 类中定义 B 类类型的属性 b、在 B 类中定义 A 类类型的属性 a。
- 一对多的关系：一个 A 类类型对应多个 B 类类型的情况，需要在 A 类中以集合的方式引入 B 类类型的对象，在 B 类中定义 A 类类型的属性 a。
- 多对多的关系：在 A 类中定义 B 类类型的集合，在 B 类中定义 A 类类型的集合。

9.2　MyBatis 中的关联关系

本节将对如何使用 MyBatis 处理对象中的 3 种关联关系进行详细讲解。

9.2.1　一对一

在现实生活中，一对一关联关系是十分常见的。例如，一个学生只有一本学生证，同时一本学生证也只对应一个学生。

那么 MyBatis 是怎么处理这种一对一关联关系的呢？在本书第 7 章所讲解的<resultMap>元素中包含一个<association>子元素，MyBatis 就是通过该元素来处理一对一关联关系的。

在<association>元素中，通常可以配置以下属性。

- property：指定映射到的实体类对象属性，与表字段一一对应。
- column：指定表中对应的字段。
- javaType：指定映射到实体对象属性的类型。
- select：指定引入嵌套查询的子 SQL 语句，用于关联映射中的嵌套查询。
- fetchType：指定在关联查询时是否启用延迟加载，有 lazy 和 eager 两个属性值，默认值为 lazy（默认关联映射延迟加载）。

<association>元素有如下两种配置方式。

```
<!--方式一：嵌套查询-->
<association property="card" column="card_id" javaType="com.ssm.po.StudentIdCard"
  select="com.ssm.mapper.StudentIdCardMapper.findCodeById"/>
<!--方式二：嵌套结果-->
<association property="card" javaType="com.ssm.po.StudentIdCard">
  <id property="id" column=""card_id"/>
  <result property="code" column="code"/>
</association>
```

注　意
MyBatis 在映射文件中加载关联关系对象主要通过两种方式：嵌套查询和嵌套结果。嵌套查询是指通过执行另一条 SQL 映射语句来返回预期的复杂类型；嵌套结果是使用嵌套结果映射来处理重复的联合结果的子集。

【示例 9-1】接下来以学生和学生证之间的一对一关联关系为例进一步进行讲解。

查询学生及其关联的学生证信息是先通过查询学生表中的主键来获取学生信息，然后通过表中的外键来获取学生证表中的学生证号信息。其具体实现步骤如下。

步骤01 创建数据表。在 db_mybatis 数据库中分别创建名为 tb_studentidcard 和 tb_student 的数据表，同时预先插入几条数据。其执行的 SQL 语句如下所示。

```
#使用数据库 db_mybatis
USE db_mybatis;
# 创建一个名称为 tb_studentidcard 的表
CREATE TABLE tb_studentidcard(
   id INT PRIMARY KEY AUTO_INCREMENT,
   CODE VARCHAR(8)
);
# 插入两条数据
INSERT INTO tb_studentidcard(CODE) VALUES('18030128');
INSERT INTo tb_studentidcard(CODE) VALUES ('18030135');
# 创建一个名称为 tb_student 的表（暂时添加少量字段）
CREATE TABLE tb_student(
   id INT PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(32),
   sex CHAR(1),
   card_id INT UNIQUE,
   FOREIGN KEY (card_id) REFERENCES tb_studentidcard(id)
);
# 插入两条数据
INSERT INTO tb_student(name, sex, card_id) VALUES('limin','f',1);
INSERT INTO tb_student(name, sex, card_id) VALUES('jack','m',2);
```

步骤02 在 Eclipse 中创建一个名为 chapter09 的 Web 项目，然后引入相关 JAR 包、MybatisUtils 工具类以及 mybatis-config.xml 核心配置文件。

步骤03 在项目的 com.ssm.po 包下创建持久化类：学生证类 StudentIdCard 和学生类 Student，编辑后的代码如文件 9.1 和文件 9.2 所示。

文件 9.1　StudentIdCard.java

```
package com.ssm.po;
//学生证类
public class StudentIdCard {
    private Integer id;
    private String code;
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getCode() {
     return code;
    }
    public void setCode(String code) {
     this.code = code;
    }
    public String toString() {
```

```
     return "StudentIdCard [id=" + id + ", code=" + code + "]";
    }
}
```

文件 9.2　Student.java

```
package com.ssm.po;
//学生类
public class Student {
    private Integer id;
    private String name;
    private String sex;
    private StudentIdCard studentIdCard;
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getName() {
     return name;
    }
    public void setName(String name) {
     this.name = name;
    }
    public String getSex() {
     return sex;
    }
    public void setSex(String sex) {
     this.sex = sex;
    }
    public StudentIdCard getStudentIdCard() {
     return studentIdCard;
    }
    public void setStudentIdCard(StudentIdCard studentIdCard) {
     this.studentIdCard = studentIdCard;
    }
    public String toString() {
     return "Student [id=" + id + ", name=" + name + ", sex=" + sex +
           ", studentIdCard=" + studentIdCard + "]";
    }
}
```

步骤04 在 com.ssm.mapper 包中创建学生证映射文件 StudentIdCardMapper.xml 和学生映射文件 StudentMapper.xml，并在两个映射文件中编写一对一关联映射查询的配置信息，如文件 9.3 和文件 9.4 所示。

文件 9.3　StudentIdCardMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.StudentIdCardMapper">
    <!--根据 id 获取学生证信息 -->
    <select id="findStudentIdCardById" parameterType="Integer" resultType="StudentIdCard">
     select * from tb_studentidcard where id=#{id}
    </select>
</mapper>
```

文件 9.4　StudentMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.StudentMapper">
    <!--嵌套查询，通过执行一条 SQL 映射语句来返回预期的特殊类型 -->
    <select id="findStudentById" parameterType="Integer"
       resultMap="StudentIdCardWithStudentResult">
     select * from tb_student where id=#{id}
    </select>
    <resultMap type="Student" id="StudentIdCardWithStudentResult">
    <id property="id" column="id"/>
       <result property="name" column="name"/>
       <result property="sex" column="sex"/>
    <!-- 一对一，association 使用 select 属性引入另一条 SQL 语句 -->
    <association property="studentIdCard" column="card_id" javaType="StudentIdCard"
        select="com.ssm.mapper.StudentIdCardMapper.findStudentIdCardById"/>
    </resultMap>
</mapper>
```

在上述两个映射文件中使用了 MyBatis 中的嵌套查询方式进行学生及其关联的学生证信息查询，因为返回的学生对象中除了基本属性外，还有一个关联的 studentIdCard 属性，所以需要手动编写结果映射。从映射文件 StudentMapper.xml 中可以看出，嵌套查询的方法是先执行一个简单的 SQL 语句，然后在进行结果映射时将关联对象在<association>元素中使用 select 属性执行另一条 SQL 语句（StudentIdCardMapper.xml 中的 SQL）。

步骤05 在核心配置文件 mybatis-config.xml 中引入 Mapper 映射文件并定义别名，如文件 9.5 所示。

文件 9.5　mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
```

```
    <!-- 引入数据库连接配置文件 -->
    <properties resource="db.properties" />
    <!-- 使用扫描包的形式定义别名 -->
    <typeAliases>
     <package name="com.ssm.po"/>
    </typeAliases>
    <environments default="mysql">
     <environment id="mysql">
         <transactionManager type="JDBC" />
         <dataSource type="POOLED">
             <!--数据库驱动 -->
             <property name="driver" value="${jdbc.driver}" />
             <!--连接数据库的 url -->
             <property name="url" value="${jdbc.url}" />
             <!--连接数据库的用户名 -->
             <property name="username" value="${jdbc.username}" />
             <!--连接数据库的密码-->
             <property name="password" value="${jdbc.password}" />
         </dataSource>
     </environment>
    </environments>
    <!-- 配置 Mapper 的位置 -->
    <mappers>
     <mapper resource="com/ssm/mapper/StudentIdCardMapper.xml" />
     <mapper resource="com/ssm/mapper/StudentMapper.xml" />
     <mapper resource="com/ssm/mapper/UserMapper.xml" />
    </mappers>
</configuration>
```

在上述核心配置文件中，首先引入了数据库连接的配置文件，然后使用扫描包的形式自定义别名，接下来进行环境的配置，最后配置了 Mapper 映射文件的位置信息。

步骤06 在 com.ssm.test 包中创建测试类 MybatisAssociatedTest，并在类中编写测试方法 findStudentByIdTest()，如文件 9.6 所示。

文件 9.6　MybatisAssociatedTest.java

```
package com.ssm.test;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;
import com.ssm.po.Student;
import com.ssm.util.MybatisUtils;
public class MybatisAssociatedTest {
    /*
     * 嵌套查询
     */
    @Test
    public void findStudentByIdTest(){
```

```
        SqlSession sqlSession = MybatisUtils.getSession();
        //使用 MyBatis 嵌套查询的方法查询 id 为 1 的学生信息
        Student student=
              sqlSession.selectOne("com.ssm.mapper.StudentMapper.findStudentById",1);
        System.out.println(student.toString());
        sqlSession.close();
    }
}
```

在文件 9.6 的 findStudentByIdTest()方法中，首先通过 MybatisUtils 工具类获取了 SqlSession 对象，然后通过 SqlSession 对象的 selectOne()方法获取了学生信息，最后关闭了 SqlSession。执行方法后，控制台的输出结果如图 9.1 所示。使用 MyBatis 嵌套查询的方式查询出了学生及其关联的学生证信息，这就是 MyBatis 中的一对一关联查询。

```
Markers  Properties  Servers  Snippets  Problems  Console  JUnit  Terminal
<terminated> MybatisAssociatedTest.findStudentByIdTest [JUnit] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年11月3日 下午8:56:31)
Student [id=1, name=limin, sex=f, studentIdCard=StudentIdCard [id=1, code=18030128]]
```

图 9.1　运行结果

虽然使用嵌套查询的方式比较简单，但是嵌套查询的方式要执行多条 SQL 语句，这对于大型数据集合和列表展示不是很好，因为这样可能会导致成百上千条关联的 SOL 语句被执行，从而极大地消耗数据库性能，并且会降低查询效率。为此，MyBatis 提供了嵌套结果的方式进行关联查询。

在 StudentMapper.xml 中，使用 MyBatis 嵌套结果的方式进行学生及其关联的学生证信息查询，所添加的代码如下所示。

```
<!--嵌套结果，通过嵌套结果映射来处理重复的联合结果的子集 -->
    <select id="findStudentById2" parameterType="Integer"
        resultMap="StudentIdCardWithStudentResult2">
        select s.*,sidcard.code
        from tb_student s,tb_studentidcard sidcard
         where s.card_id=sidcard.id and s.id=#{id}
    </select>
    <resultMap type="Student" id="StudentIdCardWithStudentResult2">
        <id property="id" column="id" />
        <result property="name" column="name" />
        <result property="sex" column="sex" />
        <association property="studentIdCard" javaType="StudentIdCard">
            <id property="id" column="card_id" />
            <result property="code" column="code" />
        </association>
    </resultMap>
```

从上述代码中可以看出，MyBatis 嵌套结果的方式只编写了一条复杂的多表关联的 SQL 语句，并且在<association>元素中继续使用相关子元素进行数据库表字段和实体类属性的一一映射。执行结果与图 9.1 相同，但使用 MyBatis 嵌套结果的方式只执行了一条 SQL 语句。

注 意

在使用 MyBatis 嵌套查询方式进行关联查询映射时，使用 MyBatis 的延迟加载在一定程度上可以降低运行消耗并提高查询效率。MyBatis 默认没有开启延迟加载，需要在核心配置文件 mybatis-config.xml 中的<settings>元素内进行配置，具体配置方式如下。

```
<settings>
<!--打开延迟加载的开关-->
<setting name="lazyLoadingEnabled" value="true"/>
<!--将积极加载改为消极加载，即按需加载-->
<setting name="aggressiveLazyLoading" value="false" />
</settings>
```

在映射文件中，MyBatis 关联映射的<association>元素和<collection>元素中都已默认配置了延迟加载属性，即默认属性 fetchType="lazy"（属性 fetchType="eager"表示立即加载），所以在配置文件中开启延迟加载后，无须在映射文件中再做配置。

9.2.2 一对多

在实际应用中，应用更多的关联关系是一对多（或多对一）。例如，一个班级有多个学生，即多个学生属于一个班级。使用 MyBatis 是怎么处理这种一对多关联关系的呢？在本书第 7 章所讲解的<resultMap>元素中包含一个<collection>子元素，MyBatis 就是通过该元素来处理一对多关联关系的。<collection>子元素的属性大部分与<collection>元素相同，但其还包含一个特殊属性——ofType。ofType 属性与 javaType 属性对应，用于指定实体对象中集合类属性所包含的元素类型。

<collection>元素可以参考如下两种示例进行配置，具体代码如下。

```
<!--方式一：嵌套查询-->
<collection property="studentList" column="id" ofType="com.ssm.po.Student"
       select= "com.ssm.mapper.StudentMapper.selectStudent"/>
<!--方式二：嵌套结果-->
<collection property="studentList" ofType="com.ssm.po.Student">
     <id property="id" column="student_id"/>
     <result property="username" column="username"/>
</collection>
```

【示例 9-2】在了解了 MyBatis 处理一对多关联关系的元素和方式后，接下来以班级和学生之间的这种一对多关联关系为例详细讲解如何在 MyBatis 中处理一对多关联关系，具体步骤如下。

步骤01 在 db_mybatis 数据库中创建两个数据表：tb_banji 和 tb_student，同时在表中预先插入几条数据，执行的 SQL 语句如下所示。

```
# 创建一个名称为 tb_banji 的表（暂添加少量字段）
CREATE TABLE tb_banji(
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(32)
```

```
);
#插入两条数据
INSERT INTO tb_banji VALUES(1,'16 软件技术 1 班');
INSERT INTO tb_banji VALUES(2,'16 软件技术 2 班');
# 创建一个名称为 tb_student 的表（暂时添加少量字段）
CREATE TABLE tb_student(
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(32),
  sex CHAR(1),
  banji_id INT ,
  FOREIGN KEY (banji_id) REFERENCES tb_banji(id)
);
#插入 3 条数据
INSERT INTO tb_student VALUES(1,'孙淼','m',1);
INSERT INTO tb_student VALUES(2,'刘梦奕','f',1);
INSERT INTO tb_student VALUES(3,'无为','m',2);
```

步骤02 在 com.ssm.po 包中创建持久化类：班级类 Banji 和学生类 Student，并在两个类中定义相关属性和方法，如文件 9.7 和文件 9.8 所示。

文件 9.7 Banji.java

```
package com.ssm.po;
import java.util.List;
//班级类
public class Banji {
    private Integer id;
    private String name;
    private List<Student> studentList;
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getName() {
     return name;
    }
    public void setName(String name) {
     this.name = name;
    }
    public List<Student> getStudentList() {
     return studentList;
    }
    public void setStudentList(List<Student> studentList) {
     this.studentList = studentList;
    }
    public String toString() {
```

```
        return "Banji [id=" + id + ", name=" + name + ", studentList=" + studentList + "]";
    }
}
```

文件 9.8　Student.java

```
package com.ssm.po;
//学生类
public class Student {
    private Integer id;
    private String name;
    private String sex;
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getName() {
     return name;
    }
    public void setName(String name) {
     this.name = name;
    }
    public String getSex() {
     return sex;
    }
    public void setSex(String sex) {
     this.sex = sex;
    }
    public String toString() {
     return "Student [id=" + id + ", name=" + name + ", sex=" + sex + "]";
    }
}
```

步骤03 在 com.ssm.mapper 包中创建班级实体映射文件 BanjiMapper.xml，并在文件中编写一对多关联映射查询的配置，如文件 9.9 所示。

文件 9.9　BanjiMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.BanjiMapper">
    <!--一对多：查看某一班级及其关联的学生信息
          注意：若关联查询出的列名相同，则需要使用别名区分 -->
    <select id="findBanjiWithStudent" parameterType="Integer"
     resultMap="BanjiWithStudentResult">
     select b.*,s.id as student_id,s.name
```

```
        from tb_banij b,tb_student s
         where b.id=s.banji_id and b.id=#{id}
       </select>
       <resultMap type="Banji" id="BanjiWithStudentResult">
        <id property="id" column="id" />
        <result property="name" column="name" />
        <!--一对多关联映射：collection
            ofType 表示属性集合中元素的类型 List<Student>属性，即 Student 类  -->
        <collection property="studentList" ofType="Student">
            <id property="id" column="student_id" />
            <result property="name" column="name" />
    <result property="sex" column="sex" />
        </collection>
       </resultMap>
    </mapper>
```

在文件 9.9 中使用 MyBatis 嵌套结果的方式定义了一个根据班级 id 查询班级及其关联的学生信息的 select 语句。因为返回的班级对象中包含 Student 集合对象属性，所以需要手动编写结果映射信息。

步骤04 将映射文件 BanjiMapper.xml 的路径配置到核心配置文件 mybatis-config.xml 中，其代码如下所示。

```
<mapper resource="com/ssm/mapper/BanjiMapper.xml" />
```

步骤05 在测试类 MyBatisAssociatedTest 中编写测试方法 findBanjiTest()。

```
@Test
public void findBanjiTest(){
     SqlSession sqlSession = MybatisUtils.getSession();
   //查询班级 id 为 1 的班级信息（及其关联的学生集合信息）
     Banji banji=
        sqlSession.selectOne("com.ssm.mapper.BanjiMapper.findBanjiWithStudent",1);
     System.out.println(banji.toString());
     sqlSession.close();
}
```

执行方法后，控制台输出结果如图 9.2 所示。使用 MyBatis 嵌套结果的方式查询出了班级及其关联的学生集合信息。这就是 MyBatis 一对多的关联查询。

```
Markers  Properties  Servers  Snippets  Problems  Console  JUnit  Terminal
<terminated> MybatisAssociatedTest.findBanjiTest [JUnit] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年11月4日 下午10:55:43)
Banji [id=1, name=16软件技术1班, studentList=[Student [id=1, name=16软件技术1班], Student [id=2, name=16软件技术1班]]
```

图 9.2　运行结果

注　意
上述案例从班级的角度出发，班级与学生之间是一对多的关联关系，但如果从单个学生的角度出发，一个学生只能属于一个班级，即一对一的关联关系。

9.2.3　多对多

在实际项目开发中，多对多的关联关系是非常常见的。以学生和课程为例，一个学生可以选修多门课程，而一门课程又可以被多个学生选修，学生和课程就属于多对多的关联关系。

在数据库中，多对多的关联关系通常使用一个中间表来维护，中间表选课表（electiveCourse）中的学生 id（student_id）作为外键参照学生表的 id，课程 id（course_id）作为外键参照课程表的 id。三个表的关联关系如图 9.3 所示。

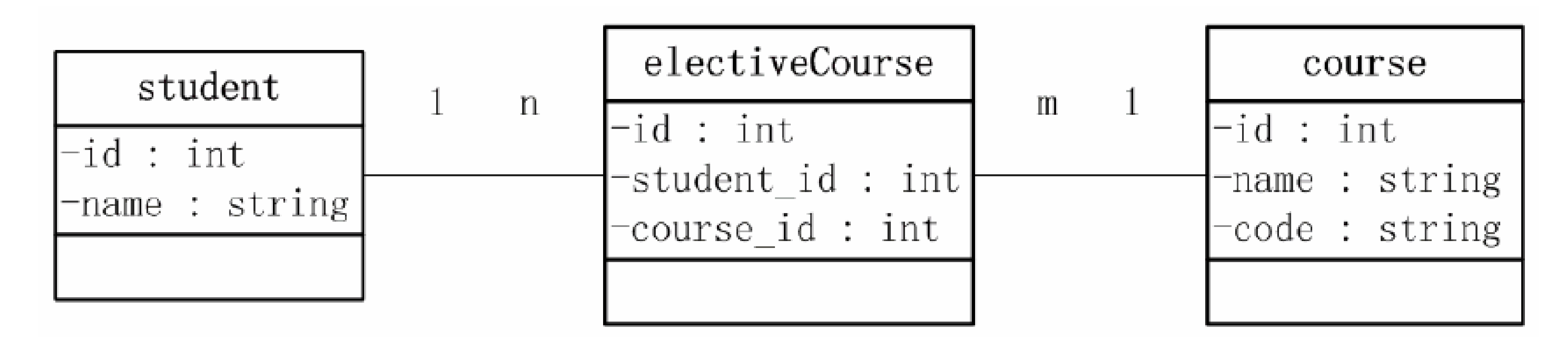


图 9.3　学生表、中间表与课程表之间的关联

【示例 9-3】了解了数据库中学生表与课程表之间的多对多关联关系后，下面我们通过具体的案例来讲解如何使用 MyBatis 处理这种多对多的关系，具体实现步骤如下。

步骤01 创建数据表。在 mybatis 数据库中新建名称为 tb_course 和 tb_electiveCourse 的两个数据表（tb_student 表已在前面的案例中创建，这里直接引用），同时在表中预先插入几条数据。其执行的 SQL 语句如下所示。

```
# 创建一个名称为 tb_course 的表
CREATE TABLE tb_course(
   id INT PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR (32),
   code VARCHAR (32)
)
# 插入两条数据
INSERT INTO tb_course VALUES (1,'Java 程序设计语言', '08113226');
INSERT INTO tb_course VALUES (2,'JavaWeb 程序开发入门','08113228');
# 创建一个名称为 tb_electiveCourse 的中间表
CREATE TABLE tb_electiveCourse (
  id INT PRIMARY KEY AUTO_INCREMENT,
  student_id INT,
  course_id INT,
  FOREIGN KEY(student_id) REFERENCES tb_student(id),
  FOREIGN KEY(course_id) REFERENCES tb_course(id)
)
# 插入 3 条数据
INSERT INTO tb_electiveCourse VALUES (1,1,1);
INSERT INTo tb_electiveCourse VALUES (2,1,2);
INSERT INTO tb_electiveCourse VALUES (3,2,2);
```

步骤02 在 com.ssm.po 包中创建持久化类课程类 Course，并在类中定义相关属性和方法，如

文件 9.10 所示。

文件 9.10　Course.java

```
package com.ssm.po;
import java.util.List;
//课程类
public class Course {
    private Integer id;
    private String name;
    private String code;
    private List<Student> studentlist;//与学生集合的关联属性
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getName() {
     return name;
    }
    public void setName(String name) {
     this.name = name;
    }
    public String getCode() {
     return code;
    }
    public void setCode(String code) {
     this.code = code;
    }
    public List<Student> getStudentlist() {
     return studentlist;
    }
    public void setStudentlist(List<Student> studentlist) {
     this.studentlist = studentlist;
    }
    public String toString() {
     return "Course [id=" + id + ", name=" + name + ", code=" + code + "]";
    }
}
```

除了需要在课程持久化类中添加学生集合的属性外，还需要在学生持久化类（Student.java）中增加课程集合的属性及其对应的 getter()/setter()方法，同时为了方便查看输出结果，需要重写 toString()。Student 类中添加的代码如下所示。

```
//关联课程集合信息
private List<Course> courseList;
//省略 getter()、setter()方法以及重写的 tostring()方法
```

步骤03 在 com.ssm.mapper 包中创建课程实体映射文件 CourseMapper.xml 和学生实体映射文件 StudentMapper.xml，对两个映射文件进行编辑后，如文件 9.11 和文件 9.12 所示。

文件 9.11　CourseMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.CourseMapper">
    <!--多对多嵌套查询：通过执行一条 SQL 映射语句来返回预期的特殊类型 -->
    <select id="findCourseWithStudent" parameterType="Integer"
     resultMap="CourseWithStudentResult">
     select * from tb_course where id=#{id}
    </select>
    <resultMap type="Course" id="CourseWithStudentResult">
     <id property="id" column="id" />
     <result property="name" column="name" />
     <result property="code" column="code" />
     <collection property="studentList" column="id" ofType="Student"
 select="com.ssm.mapper.StudentMapper.findStudentById">
     </collection>
    </resultMap>
</mapper>
```

在文件 9.11 中，使用嵌套查询的方式定义了一个 id 为 findCourseWithStudent 的 select 语句来查询课程及其关联的学生信息。在<resultMap>元素中使用了<collection>元素来映射多对多的关联关系，其中 property 属性表示订单持久化类中的课程属性，ofType 属性表示集合中的数据为 Student 类型，而 column 的属性值会作为参数执行 StudentMapper.xml 中定义的 id 为 findStudentById 的执行语句来查询订单中的学生信息。

文件 9.12　StudentMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.StudentMapper">
    <select id="findStudentById" parameterType="Integer" resultType="Student">
     select * from tb_student where id in(
     select student_id from tb_electivecourse where course_id=#{id}
     )
    </select>
</mapper>
```

在文件 9.12 中定义了一个 id 为 findStudentById 的执行语句，该执行语句中的 SQL 会根据课程 id 查询与该课程所关联的学生信息。由于课程和学生是多对多的关联关系，因此需要通过中间表来查询学生信息。

步骤04 将新创建的映射文件 CourseMapper.xml 和 StudentMapper.xml 的文件路径配置到核心

配置文件 mybatis-config.xml 中，代码如下所示。

```
<mapper resource="com/ssm/mapper/CourseMapper.xml" />
<mapper resource="com/ssm/mapper/StudentMapper.xml" />
```

步骤05 在测试类 MyBatisAssociatedTest 中编写多对多关联查询的测试方法 findCourseByIdTest()，其代码如下所示。

```
/*
* 多对多嵌套查询
*/
@Test
public void findCourseByIdTest(){
    SqlSession sqlSession = MybatisUtils.getSession();
    //查询课程 id 为 1 的课程中的学生信息
    Course course=
     sqlSession.selectOne("com.ssm.mapper.CourseMapper.findCourseWithStudent", 1);
    System.out.println(course);
    sqlSession.close();
}
```

执行方法后，控制台的输出结果如图 9.4 所示。使用 MyBatis 嵌套查询的方式查询出了课程及其关联的学生信息，这就是 MyBatis 多对多的关联查询。

```
<terminated> MybatisAssociatedTest.findCourseByIdTest [JUnit] C:\Program Files\Java\jdk1.7.0_72\bin\javaw.exe (2018年11月5日 下午3:57:01)
Course [id=1, name=Java, code=08113226, studentList=[Student [id=1, name=jack], Student [id=2, name=rose]]]
```

图 9.4 运行结果

如果读者对多表关联查询的 SQL 语句比较熟，就可以在 CourseMapper.xml 中使用嵌套结果的方式，其代码如下所示。

```
<!--多对多嵌套结果查询：查询某课程及其关联的学生详情 -->
    <select id="findCourseWithStudent2" parameterType="Integer"
        resultMap="CourseWithStudentResult2">
        select c.*,s.id as sid,s.name
        from tb_course c,tb_student s,tb_electivecourse ec
        where ec.course_id=c.id
        and ec.student_id=s.id and c.id=#{id}
    </select>
    <resultMap type="Course" id="CourseWithStudentResult2">
        <id property="id" column="id" />
        <result property="name" column="name" />
        <result property="code" column="code" />
        <collection property="studentList" ofType="Student">
            <id property="id" column="sid" />
            <result property="name" column="name" />
        </collection>
```

```
    </resultMap>
```

9.3　习　　题

1. 请简述不同对象之间的 3 种关联关系。
2. 针对不同对象之间的 3 种关联关系，列举现实中存在的案例，并进行编码实践。

第 10 章

MyBatis 与 Spring 的整合

前面章节分别讲解了 Spring 和 MyBatis 的相关知识，在实际的项目开发中，需要将 Spring 与 MyBatis 整合在一起使用。

本章主要涉及的知识点如下：

- 整合环境搭建。
- 传统 DAO 方式的开发整合。
- Mapper 接口方式的开发整合。

10.1 整合环境搭建

Spring 和 MyBatis 的整合主要涉及准备所需的 JAR 包和编写配置文件，下面将详细介绍。

10.1.1 准备所需的 JAR 包

要实现 MyBatis 与 Spring 的整合，需要这两个框架相关的 JAR 包，除此之外，还需要其他的 JAR 包来配合使用。

1. 所需 Spring 框架的 JAR 包

Spring 框架所需要准备的 JAR 包共 10 个，其中包括 4 个核心模块 JAR、AOP 开发使用的 JAR、JDBC 和事务的 JAR（其中核心容器依赖的 commons-logging 的 JAR 在 MyBatis 框架的 lib 包中已经包含），具体如下所示。

- aopalliance-1.0.jar

- aspectjweaver-1.8.10.jar
- spring-aop-4.3.6.RELEASE.jar
- spring-aspects-4.3.6.RELEASE.jar
- spring-beans-4.3.6.RELEASE.jar
- spring-context-4,3.6.RELEASE.jar
- spring-core-4.3.6.RELEASE.jar
- spring-expression-4.3.6.RELEASE.jar
- spring-jdbc-4.3.6.RELEASE.jar
- spring-tx-4.3.6.RELEASE.jar

2. 所需 MyBatis 框架的 JAR 包

MyBatis 框架需要准备的 JAR 包共 13 个，其中包括核心包 mybatis-3.4.2.jar 及其解压文件夹中 lib 目录中的所有 JAR，具体如下所示。

- ant-1.9.6.jar
- ant-launcher-1.9.6.jar
- asm-5.1.jar
- cglib-3.2.4.jar
- commons-logging-1.2.jar
- javassist-3.21.0-GA.jar
- log4j-1.2.17.jar
- log4j-api-2.3ar
- log4j-core-2.3.jar
- mybatis-3.4.2.jar
- ognl-3.1.12.jar
- slf4j-api-1.7.22.jar
- slf4j-log4j12-1.7.22.jar

3. MyBatis 与 Spring 整合所需的中间 JAR 包

为了满足 MyBatis 用户对 Spring 框架的需求，MyBatis 社区开发了一个用于整合 MyBatis 和 Spring 两个框架的中间件——MyBatis-Spring。

本书使用的中间件是 mybatis-spring-1.3.1jar。此版本的 JAR 包可以通过链接获取 https://mvnrepository.com/artifact/org.mybatis/mybatis-spring/1.3.1。

4. 数据库驱动 JAR 包

本书所使用的数据库驱动包为 mysql-connector-java-5.1.7-bin.jar。

5. 数据源所需 JAR 包

整合时所使用的是 DBCP 数据源，所以需要准备 DBCP 和连接池的 JAR 包，具体如下所示。

- commons-dbcp2-2.1.1.jar

- commons-pool2-2.4.2.jar

10.1.2 编写配置文件

在 Eclipse 中创建一个名称为 chapter10 的 Web 项目，将 10.1.1 小节准备的全部 JAR 包添加到项目的 lib 目录中，并发布到类路径下。

【示例 10-1】参照前面章节的内容和案例，在项目的 src 目录下分别创建 db.properties 文件、Spring 的配置文件 applicationContext.xml 以及 MyBatis 的配置文件 mybatis-config.xml，如文件 10.1、10.2 和 10.3 所示。

文件 10.1 db.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/db_mybatis
jdbc.username=root
jdbc.password=root
jdbc.maxTotal=30
jdbc.maxIdle=10
jdbc.initialSize=5
```

在文件 10.1 中，除了配置连接数据库的基本 4 项外，还配置数据库连接池的最大连接数（maxTotal）、最大空闲连接数（maxIdle）以及初始化连接数（initialSize）。

文件 10.2 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!--读取 db.properties-->
    <context:property-placeholder location="classpath:db.properties"/>
    <!--配置数据源 -->
    <bean id="dataSource"
     class="org.apache.commons.dbcp2.BasicDataSource">
     <!--数据库驱动 -->
     <property name="driverClassName" value="${jdbc.driver}" />
     <!--连接数据库的 url -->
```

```
        <property name="url" value="${jdbc.url}" />
        <!--连接数据库的用户名 -->
        <property name="username" value="${jdbc.username}" />
        <!--连接数据库的密码-->
        <property name="password" value="${jdbc.password}" />
        <!--最大连接数-->
        <property name="maxTotal" value="${jdbc.maxTotal}" />
        <!--最大空闲连接-->
        <property name="maxIdle" value="${jdbc.maxIdle}" />
        <!--初始化连接数-->
        <property name="initialSize" value="${jdbc.initialSize}" />
    </bean>
    <!--事务管理器，依赖于数据源 -->
    <bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!--注册事务管理器驱动，开启事务注解 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>
    <!--配置 MyBatis 工厂 -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--注入数据源 -->
        <property name="dataSource" ref="dataSource" />
        <!--指定核心配置文件位置 -->
        <property name="configLocation" value="classpath:mybatis-config.xml" />
    </bean>
    <bean id="userDao" class="com.ssm.dao.impl.UserDaoImpl">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
    </bean>
</beans>
```

在文件 10.2 中，首先定义了读取 properties 文件的配置，然后配置了数据源，接下来配置了事务管理器并开启了事务注解，最后配置了 MyBatis 工厂来与 Spring 整合。其中，MyBatis 工厂的作用是构建 SqlSessionFactory，它是通过 MyBatis-Spring 包中提供的 org.mybatis.Spring.SqlSessionFactoryBean 类来配置的。通常在配置时需要提供两个参数：一个是数据源；另一个是 MyBatis 的配置文件路径。这样 Spring 的 IoC 容器就会在初始化 id 为 sqlSessionFactory 的 Bean 时解析 MyBatis 的配置文件，并与数据源一同保存到 Spring 的 Bean 中。

文件 10.3　mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--配置别名 -->
    <typeAliases>
        <package name="com.ssm.po"/>
```

```
    </typeAliases>
    <!--配置 Mapper 的位置 -->
    <mappers>
     ...
    </mappers>
</configuration>
```

由于在 Spring 中已经配置了数据源信息，因此在 MyBatis 的配置文件中不再需要配置数据源信息。这里只需要使用<typeAliases>和<mappers>元素来配置文件别名以及指定 mapper 文件位置即可。

此外，还需在项目的 src 目录下创建 log4j.properties 文件，该文件的编写可参考第 6 章的入门案例，也可将前面章节创建的该文件复制到此项目中使用。

10.2 整　　合

10.1 节已经完成了对 MyBatis 与 Spring 整合环境的搭建工作，可以说完成这些配置后就已经完成了这两个框架大部分的整合工作。接下来，本节将对通过传统 DAO 方式和 Mapper 接口方式的开发整合进行介绍。

10.2.1 传统 DAO 方式的开发整合

采用传统 DAO 开发方式进行 MyBatis 与 Spring 框架的整合时，我们需要编写 DAO 接口以及接口的实现类，并且需要向 DAO 实现类中注入 SqlSessionFactory，然后在方法体内通过 SqlSessionFactory 创建 SqlSession。为此，我们可以使用 Mybatis-Spring 包中所提供的 SqlSessionTemplate 类或 SqlSessionDaoSupport 类来实现此功能。这两个类的描述如下。

- SqlSessionTemplate：Mybatis-Spring 的核心类，它负责管理 MyBatis 的 SqlSession，调用 MyBatis 的 SQL 方法。当调用 SQL 方法时，SqlSessionTemplate 将会保证使用的 SqlSession 和当前 Spring 的事务是相关的。它还管理 SqlSession 的生命周期，包含必要的关闭、提交和回滚操作。
- SqlSessionDaoSupport：一个抽象支持类，它继承了 DaoSupport 类，主要是作为 DAO 的基类来使用。可以通过 SqlSessionDaoSupport 类的 getSqlSession()方法来获取所需的 SqlSession。

【示例 10-2】了解了传统 DAO 开发方式整合可以使用的两个类后，下面以 SqlSessionDaoSupport 类的使用为例讲解传统的 DAO 开发方式整合的实现，其具体步骤如下。

1. 实现持久层

步骤01 在 src 目录下创建一个 com.ssm.po 包，并在包中创建持久化类用户类 User，在 User 类中定义相关属性和方法，如文件 10.4 所示。

文件 10.4　User.java

```
package com.ssm.po;
```

```
//用户类
public class User {
    private Integer id;
    private String username;
    private String jobs;
    private String phone;
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getUsername() {
     return username;
    }
    public void setUsername(String username) {
     this.username = username;
    }
    public String getJobs() {
     return jobs;
    }
    public void setJobs(String jobs) {
     this.jobs = jobs;
    }
    public String getPhone() {
     return phone;
    }
    public void setPhone(String phone) {
     this.phone = phone;
    }
    public String toString() {
     return "User [id=" + id + ", username=" + username + ", jobs=" + jobs +
", phone=" + phone + "]";
    }
}
```

步骤02 在 com.ssm.po 包中创建映射文件 UserMapper.xml，在该文件中编写根据 id 查询用户信息的映射语句，如文件 10.5 所示。

文件 10.5　UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.po.UserMapper">
    <!--根据用户编号获取用户信息 -->
    <select id="findUserById" parameterType="Integer" resultType="User">
     select * from t_user where id=#{id}
```

```
    </select>
</mapper>
```

步骤03 在 MyBatis 的配置文件 mybatis-config.xml 中配置映射文件 UserMapper.xml 的位置，具体如下。

```
<mapper resource="com/ssm/po/UserMapper.xml" />
```

2. 实现 DAO 层

步骤01 在 src 目录下创建一个 com.ssm.dao 包，并在包中创建接口 UseDao，在接口中编写一个通过 id 查询用户的方法 findUserById()，如文件 10.6 所示。

文件 10.6 UserDao.iava

```
package com.ssm.dao;
import com.ssm.po.User;
//用户接口类
public interface UserDao {
 //根据用户 id 查询用户的方法
    public User findUserById(Integer id);
}
```

步骤02 在 src 目录下创建一个 com.ssm.dao.impl 包，并在包中创建 UserDao 接口的实现类 UserDaoImpl，如文件 10.7 所示。

文件 10.7 UserDaoImpl.java

```
package com.ssm.dao.impl;
import org.mybatis.spring.support.SqlSessionDaoSupport;
import com.ssm.dao.UserDao;
import com.ssm.po.User;
//用户接口实现类
public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {
 //根据用户 id 查询用户的实现方法
    public User findUserById(Integer id) {
     return this.getSqlSession().selectOne("com.ssm.po.UserMapper.findUserById", id);
    }
}
```

在文件 10.7 中，UserDaoImpl 类继承了 SqlSessionDaoSupport 并实现了 UserDao 接口。其中，SqlSessionDaoSupport 类在使用时需要一个 SqlSessionFactory 或一个 SqlSessionTemplate 对象，所以需要通过 Spring 给 SqlSessionDaoSupport 类的子类对象注入一个 SqlSessionFactory 或 SqlSessionTemplate。这样，子类中就能通过调用 SqlSessionDaoSupport 类的 getSqlSession()方法来获取 SqlSession 对象，并使用 SqlSession 对象中的方法了。

步骤03 在 Spring 的配置文件 applicationContext.xml 中编写实例化 UserDaoImpl 的配置，代码如下所示。

```
<!-- 实例化 Dao -->
```

```
<bean id="userDao" class="com.ssm.dao.impl.UserDaoImpl">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>
```

上述代码创建了一个 id 为 userDao 的 Bean，并将 SqlSessionFactory 对象注入该 Bean 的实例化对象中。

3. 整合测试

在 src 目录下创建一个 com.ssm.test 包，在包中创建测试类 UserDaoTest，并在类中编写测试方法 findUserByIdDaoTest()，如文件 10.8 所示。

文件 10.8　DaoTest.java

```java
package com.ssm.test;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.ssm.dao.UserDao;
import com.ssm.po.User;
public class UserDaoTest {
    @Test
    public void findUserByIdDaotest(){
     //1.初始化 Spring 容器，加载配置文件
     ApplicationContext applicationContext=
new ClassPathXmlApplicationContext("applicationContext.xml");
     //2.通过容器获取 userDao 实例
     UserDao userDao=(UserDao)applicationContext.getBean("userDao");
     //调用 UserDao 接口的查询用户方法（用户 id 值为 1）
     User user=userDao.findUserById(1);
     System.out.println(user);
    }
}
```

在上述方法中，我们采用的是根据容器中 Bean 的 id 来获取指定 Bean 的方式。执行上述方法后，控制台的输出结果如图 10.1 所示。从中可以看出，通过 UserDao 实例的 findUserById()方法已经查询出了 id 为 1 的用户信息，这就说明 MyBatis 与 Spring 整合成功。

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Problems  Console  Progress  JUnit
<terminated> UserDaoTest.findUserByIdDaotest [JUnit] C:\Program Files\Java\jdk1.7.0_72\bin\javaw.exe (2018年11月6日 下午4:28:54)
User [id=1, username=zhangsan, jobs=teacher, phone=13907998372]
```

图 10.1　运行结果

10.2.2　Mapper 接口方式的开发整合

在 MyBatis+Spring 的项目中，虽然使用传统的 DAO 开发方式可以实现所需功能，但是采用这种方式在实现类中会出现大量的重复代码，在方法中也需要指定映射文件中执行语句的 id，并且不能保证编写时 id 的正确性（运行时才能知道）。为此，我们可以使用 MyBatis 提供的另一种编程方

式，即使用 Mapper 接口编程。接下来将讲解如何使用 Mapper 接口方式来实现 MyBatis 与 Spring 的整合。

1. 基于 MapperFactoryBean 的整合

MapperFactoryBean 是 MyBatis-Spring 团队提供的一个用于根据 Mapper 接口生成 Mapper 对象的类，该类在 Spring 配置文件中使用时可以配置以下参数。

- mapperInterface：用于指定接口。
- SqlSessionFactory：用于指定 SqlSessionFactory。
- SqlSessionTemplate：用于指定 SqlSessionTemplate。若与 SqlSessionFactory 同时设定，则只会启用 SqlsessionTemplate。

【示例 10-3】了解了 MapperFactoryBean 类后，接下来通过一个具体的案例来演示如何通过 MapperFactoryBean 实现 MyBatis 与 Spring 的整合，具体步骤如下。

步骤01 在 src 目录下创建一个 com.ssm.mapper 包，然后在该包中创建 UserMapper 接口以及对应的映射文件，如文件 10.9 和文件 10.10 所示。

文件 10.9 UserMapper.java

```java
package com.ssm.mapper;
import com.ssm.po.User;
public interface UserMapper {
    public User findUserById(Integer id);
}
```

文件 10.10 UserMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.UserMapper">
    <!--根据用户编号获取用户信息 -->
    <select id="findUserById" parameterType="Integer" resultType="User">
     select * from t_user where id=#{id}
    </select>
</mapper>
```

从文件 10.9 和文件 10.10 可以看出，这两个文件的实现代码与 10.2.1 小节中 UserDao 接口以及映射文件 UserMapper.xml 的实现代码基本相同，只是本案例与 10.2.1 小节的案例的区别在于，本案例将接口文件改名并与映射文件一起放在了 com.ssm.mapper 包中。

步骤02 在 MyBatis 的配置文件中引入新的映射文件，代码如下所示。

```xml
<!-- Mapper 接口开发方式 -->
<mapper resource="com/ssm/mapper/UserMapper.xml" />
```

步骤03 在 Spring 的配置文件中创建一个 id 为 userMapper 的 Bean，代码如下所示。

```
<!-- Mapper 代理开发（基于 MapperFactoryBean） -->
<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="com.ssm.mapper.UserMapper" />
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

上述配置代码为 MapperFactoryBean 指定了接口以及 SqlSessionFactory。

步骤04 在测试类 UserDaoTest 中编写测试方法 findUserByIdMapperTest()，其代码如下所示。

```
@Test
public void findUserByIdMapperTest(){
    ApplicationContext applicationContext=
                new ClassPathXmlApplicationContext("applicationContext.xml");
    UserMapper userMapper=(UserMapper)applicationContext.getBean("userMapper");
    User user=userMapper.findUserById(1);
    System.out.println(user);
}
```

上述方法中，通过 Spring 容器获取了 UserMapper 实例，并调用了实例中的 findUserById()方法来查询 id 为 1 的用户信息。执行方法后，控制台的输出结果与图 10.1 相同。

注　意

Mapper 接口编程方式是，只需要程序员编写 Mapper 接口（相当于 DAO 接口），然后由 MyBatis 框架根据接口的定义创建接口的动态代理对象，这个代理对象的方法体等同于 10.2.1 小节中 DAO 接口的实现类方法。

虽然使用 Mapper 接口编程的方式很简单，但是在具体使用时还是需要遵循以下规范。

（1）Mapper 接口的名称和对应的 Mapper.xml 映射文件的名称必须一致。

（2）Mapper.xml 文件中的 namespace 与 Mapper 接口的类路径相同（即接口文件和映射文件需要放在同一个包中）。

（3）Mapper 接口中的方法名和 Mapper.xml 中定义的每个执行语句的 id 相同。

（4）Mapper 接口中方法的输入参数类型要和 Mapper.xml 中定义的每个 SQL 的 parameterType 的类型相同。

（5）Mapper 接口方法的输出参数类型要和 Mapper.xml 中定义的每个 SQL 的 resultType 的类型相同。

只要遵循了这些开发规范，MyBatis 就可以自动生成 Mapper 接口实现类的代理对象，从而简化开发。

2. 基于 MapperScannerConfigurer 的整合

在实际的项目中，DAO 层会包含很多接口，如果每一个接口都像 10.2.1 小节中那样在 Spring 配置文件中配置，那么不但会增加工作量，还会使得 Spring 配置文件非常臃肿。为此，MyBatis-Spring 团队提供了一种自动扫描的形式来配置 MyBatis 中的映射器——采用 MapperScannerConfigurer 类。

MapperScannerConfigurer 类在 Spring 配置文件中使用时可以配置以下属性。

- basePackage：指定映射接口文件所在的包路径，当需要扫描多个包时可以使用分号或逗号作为分隔符。指定包路径后，会扫描该包及其子包中的所有文件。
- annotationClass：指定要扫描的注解名称，只有被注解标识的类才会被配置为映射器。
- sqlSessionFactoryBeanName：指定在 Spring 中定义的 SqlSessionFactory 的 Bean 名称。
- sqlSessionTemplateBeanName：指定在 Spring 中定义的 SqlSessionTemplate 的 Bean 名称。若定义此属性，则 sqlSessionFactoryBeanName 将不起作用。
- markerInterface：指定创建映射器的接口。

MapperScannerConfigurer 的使用非常简单，只需要在 Spring 的配置文件中编写如下代码：

```
<!-- Mapper 代理开发（基于 MapperScannerConfigurer） -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.ssm.mapper" />
</bean>
```

在通常情况下，MapperScannerConfigurer 在使用时只需通过 basePackage 属性指定需要扫描的包即可。Spring 会自动地通过包中的接口生成映射器。这使得开发人员可以在编写很少代码的情况下完成对映射器的配置，从而提高开发效率。

要验证上面的配置很容易，只需将上述配置代码写入 Spring 的配置文件，并将基于 MapperFactoryBean 的整合案例中的步骤 02 和步骤 03 的代码注释掉，再次执行 findUserByIdMapperTest()方法进行测试即可。方法执行后结果一致。

10.3 习 题

1. 简述 MyBatis 与 Spring 框架整合所需要的 JAR 包。
2. 动手搭建对 MyBatis 与 Spring 框架整合的环境，并分别使用传统 DAO 方式和基于 Mapper 接口的方式进行开发整合实践。

第 11 章

Spring MVC 入门

在学习了 Spring 框架和 MyBatis 框架的使用及整合后，接下来将讲解 Spring MVC 框架。

本章主要涉及的知识点如下：

- Spring MVC 的特点。
- Spring MVC 入门程序。
- Spring MVC 核心类。
- Spring MVC 注解。

11.1 Spring MVC 概述

Spring MVC 是 Spring 提供的一个轻量级 Web 框架，它实现了 Web MVC 设计模式。Spring MVC 在使用和性能等方面比另一个框架 Struts 2 更加优异。

Spring MVC 具有如下特点。

- 是 Spring 框架的一部分，可以方便地利用 Spring 所提供的其他功能。
- 灵活性强，易于与其他框架集成。
- 提供了一个前端控制器 DispatcherServlet，使开发人员无须额外开发控制器对象。
- 可自动绑定用户输入，并能正确地转换数据类型。
- 内置了常见的校验器，可以校验用户输入。如果校验不能通过，就会复位向到输入表单。
- 支持国际化，可以根据用户区域显示多国语言。
- 支持多种视图技术，如 JSP、Velocity 和 FreeMarker 等视图技术。
- 使用基于 XML 的配置文件，在编辑后，不需要重新编译应用程序。

11.2 案例——第一个 Spring MVC 应用

了解了什么是 Spring MVC 及其优点后，接下来通过一个简单的入门案例来演示 Spring MVC 的使用。

11.2.1 创建项目，引入 JAR 包

在 Eclipse 中创建一个名称为 chapter11 的 Web 项目，在项目的 lib 目录中添加运行 Spring MVC 程序所需要的 JAR 包，并发布到类路径下，如图 11.1 所示。

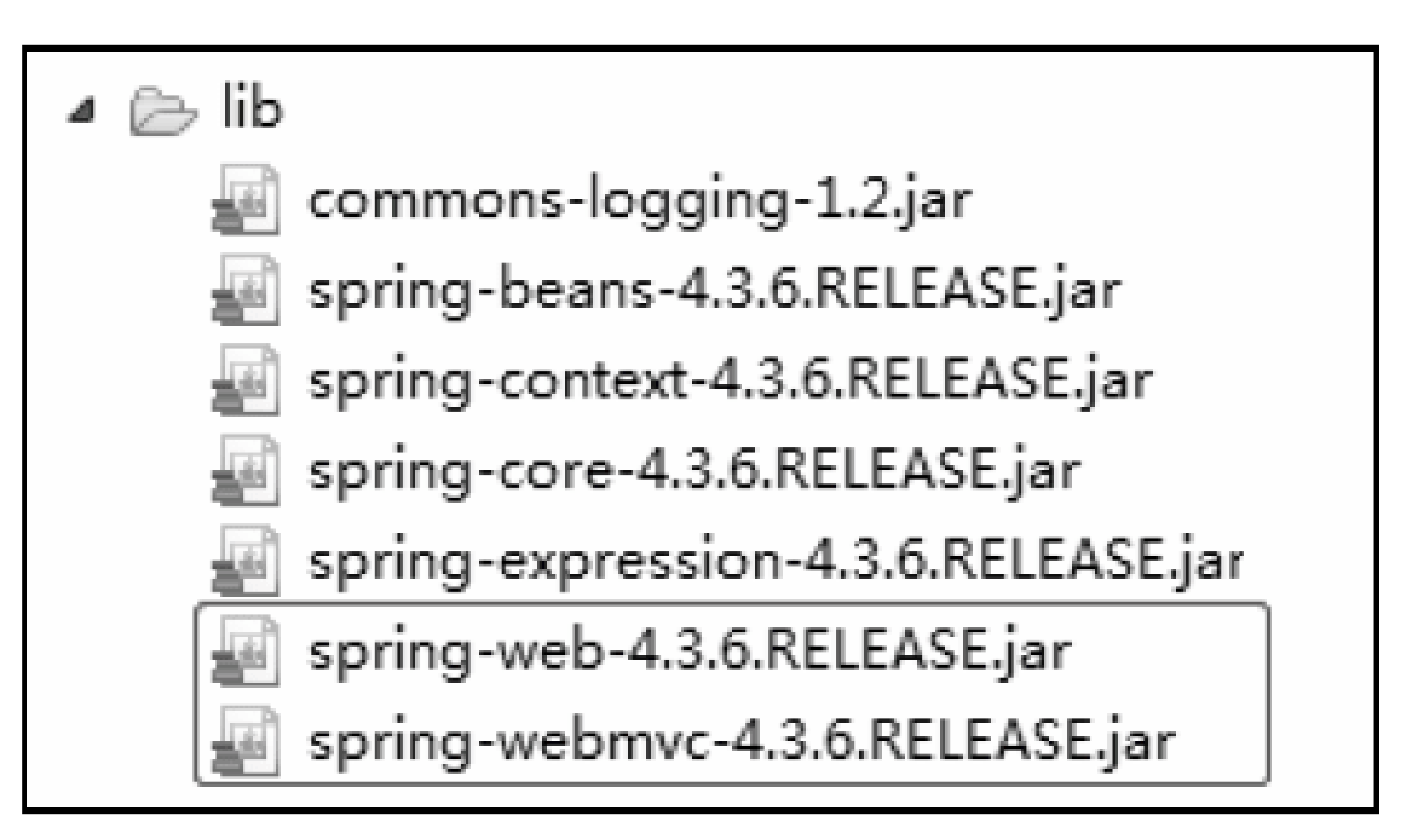


图 11.1　所需要的 JAR 包

从图 11.1 可以看到，项目中添加了 4 个核心 JAR 包、commons-logging 的 JAR 包以及两个 Web 相关的 JAR 包，这两个 Web 相关的 JAR 包就是 Spring MVC 框架所需的 JAR 包。

11.2.2 配置前端控制器

在 web.xml 中配置 Spring MVC 的前端控制器 DispatcherServlet，如文件 11.1 所示。

文件 11.1　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
        xmlns=http://java.sun.com/xml/ns/javaee
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        id="WebApp_ID" version="3.0">
  <servlet>
  <!-- 配置前端过滤器 -->
      <servlet-name>springmvc</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <!-- 初始化时加载配置文件 -->
      <init-param>
          <param-name>contextConfigLocation</param-name>
```

```
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <!-- 表示容器在启动时立即加载 Servlet -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

在文件 11.1 中，主要对<servlet>和<servlet- mapping>元素进行了配置。在<servlet>中配置了 Spring MVC 的前端控制器 DispatcherServlet，并通过其子元素<init-param>配置了 Spring MVC 配置文件的位置，<load-on-startup>元素中的 1 表示容器在启动时立即加载这个 Servlet；在<servlet-mapping>中，通过<url-pattern>元素的“/”拦截所有 URL，并交由 DispatcherServlet 处理。

11.2.3 创建 Controller 类

在 src 目录下创建一个 com.ssm.controller 包，并在包中创建控制器类 ControllerTest，该类需要实现 Controller 接口，如文件 11.2 所示。

文件 11.2 ControllerTest.java

```
package com.ssm.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
public class ControllerTest implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest arg0, HttpServletResponse arg1)
                                                    throws Exception {
     //创建 ModelAndView 对象
     ModelAndView m=new ModelAndView();
     //向模型对象中添加一个名称为 msg 的字符串对象
     m.addObject("msg","第一个 Spring MVC 程序");
     //设置返回的视图路径
     m.setViewName("/WEB-INF/jsp/welcome.jsp");
     return m;
    }
}
```

在文件 11.2 中，handleRequest()是 Controller 接口的实现方法，ControllerTest 类会调用该方法处理请求，并返回一个包含视图名或包含视图名与模型的 ModelAndview 对象。本案例向模型对象中添加了一个名称为 msg 的字符串对象，并设置返回的视图路径为 WEB-INF/jsp/welcome. jsp，这样请求就会被转发到 welcome. jsp 页面。

11.2.4 创建 Spring MVC 的配置文件，配置控制器映射信息

在 src 目录下创建配置文件 springmvc-config.xml，并在文件中配置控制器信息，如文件 11.3 所示。

文件 11.3 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
<!--配置处理器 Handle，映射"controllerTest"请求 -->
<bean name="/controllerTest" class="com.ssm.controller.ControllerTest" />
<!--处理器映射，将处理器 Handle 的 name 作为 url 进行查找 -->
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
<!--处理器适配器，配置对处理器中 handleRequest()方法的调用 -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />
<!--视图解析器 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" />
</beans>
```

在文件 11.3 中，首先定义了一个名称为“/ControllerTest”的 Bean，该 Bean 会将控制器类 ControllerTest 映射到“/ControllerTest”请求中；然后配置了处理器映射器 BeanNameUrlHandlerMapping 和处理器适配器 SimpleControllerHandlerAdapter，其中处理器映射器用于将处理器 Bean 中的的 name（即 url）进行处理器查找，而处理器适配器用于完成对 ControllerTest 处理器中 handleRequest()方法的调用；最后配置了视图解析器 InternalResourceViewResolver 来解析结果视图，并将结果呈现给用户。

> **注 意**
>
> 在老版本的 Spring 中，配置文件内必须要配置处理器映射器、处理器适配器和视图解析器。但在 Spring 4.0 以后，如果不配置处理器映射器、处理器适配器和视图解析器，就会使用 Spring 内部默认的配置来完成相应的工作，这里显示的配置处理器映射器、处理器适配器和视图解析器是为了让读者能够更加清晰地了解 Spring MVC 的工作流程。

11.2.5 创建视图（View）页面

在 WEB-INF 目录下创建一个 JSP 文件夹，并在文件夹中创建一个页面文件 welcome.jsp，在该页面中使用 EL 表达式获取 msg 中的信息，如文件 11.4 所示。

文件 11.4 first. jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>入门程序</title>
    </head>
    <body>
        ${msg}
    </body>
</html>
```

11.2.6　启动项目，测试应用

将 chapter11 项目发布到 Tomcat 服务器并启动服务器。在浏览器中访问 http://localhost:8080/chapter11/controllerTest，显示的页面效果如图 11.2 所示。从中可以看到，浏览器中已经显示出了模型对象中的字符串信息，这就说明第一个 Spring MVC 程序执行成功。

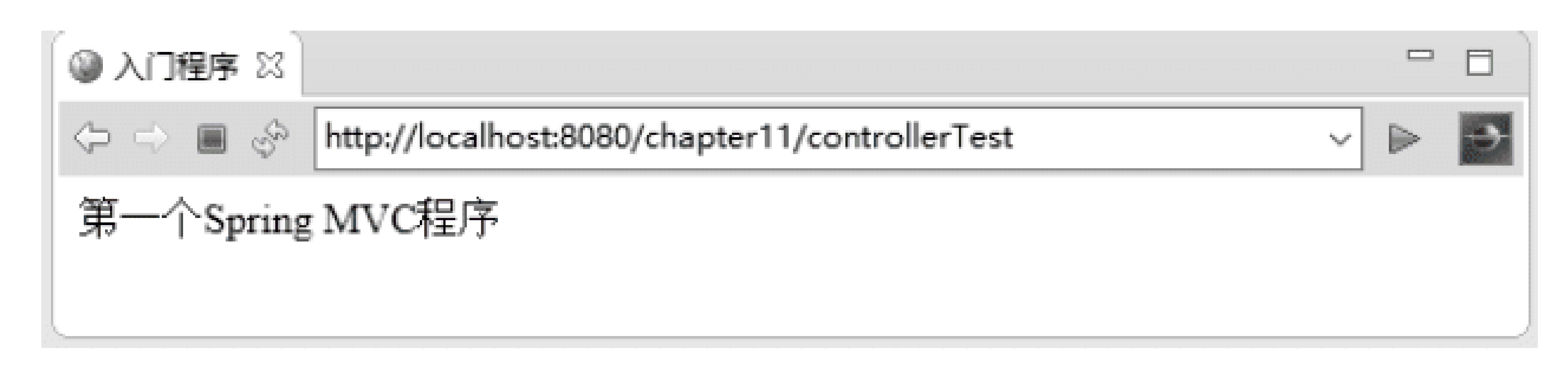


图 11.2　运行结果页面效果

通过入门案例的学习，可以总结一下 Spring MVC 程序的执行流程。

（1）用户通过浏览器向服务器发送请求，请求会被 Spring MVC 的前端控制器 DispatcherServlet 所拦截。

（2）DispatcherServlet 拦截到请求后，会调用 HandlerMapping 处理器映射器。

（3）处理器映射器根据请求 URL 找到具体的处理器，生成处理器对象及处理器拦截器（如果有就生成）一并返回给 DispatcherServlet。

（4）DispatcherServlet 会通过返回信息选择合适的 HandlerAdapter（处理器适配器）。

（5）HandlerAdapter 会调用并执行 Handler（处理器），这里的处理器就是程序中编写的 Controller 类，也被称为后端控制器。

（6）Controller 执行完成后，会返回一个 ModelAndView 对象，该对象中包含视图名或包含模型与视图名。

（7）HandlerAdapter 将 ModelAndView 对象返回给 DispatcherServlet。

（8）DispatcherServlet 会根据 ModelAndView 对象选择一个合适的 ViewResolver（视图解析器）。

（9）ViewResolver 解析后，会向 DispatcherServlet 中返回具体的 View（视图）。

（10）DispatcherServlet 对 View 进行渲染（即将模型数据填充至视图中）。

（11）视图渲染结果会返回给客户端浏览器显示。

在上述执行过程中，DispatcherServlet、HandlerMapping、HandlerAdapter 和 ViewResolver 对象的工作是在框架内部执行的，开发人员并不需要关心这些对象内部的实现过程，只需要配置前端控

制器（DispatcherServlet），完成 Controller 中的业务处理，并在视图中（View）中展示相应信息即可。

11.3 Spring MVC 的注解

在 Spring 2.5 之前，只能使用实现 Controller 接口的方式来开发一个控制器，11.2 节的案例就是使用的这种方式。在 Spring 2.5 之后，新增加了基于注解的控制器以及其他一些常用注解，这些注解的使用极大地减少了程序员的开发工作。本节将详细讲解 Spring MVC 中的常用核心类及其常用注解。

11.3.1 DispatcherServlet

DispatcherServlet 的全名是 org.Springframework.web.servlet.DispatcherServlet，它在程序中充当着前端控制器的角色。

【示例 11-1】在使用 DispatcherServlet 时，只需将其配置在项目的 web.xml 文件中，其配置代码如下。

```
<servlet>
 <!-- 配置前端过滤器 -->
 <servlet-name>springmvc</servlet-name>
 <servlet-class>org.srpingframework.web.servlet.DispatcherServlet</servlet-class>
 <!-- 初始化时加载配置文件 -->
 <init-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>classpath:springmvc-config.xml</param-value>
 </init-param>
 <!-- 表示容器在启动时立即加载 Servlet -->
 <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
 <servlet-name>springmvc</servlet-name>
 <url-pattern>/</url-pattern>
</servlet-mapping>
```

在上述代码中，<load-on-startup>元素和<init-param>元素都是可选的。如果<load-on-startup>元素的值为 1，在应用程序启动时就会立即加载该 Servlet；如果<load-on-startup>元素不存在，应用程序会就在第一个 Servlet 请求时加载该 Servlet。如果<init-param>元素存在并且通过其子元素配置了 Spring MVC 配置文件的路径，应用程序在启动时就会加载配置路径下的配置文件；如果没有通过<init-param>元素配置，应用程序就会默认到 WEB-INF 目录下寻找如下方式命名的配置文件。

```
servletName-servlet.xml
```

其中，servletName 指的是部署在 web.xml 中的 DispatcherServlet 的名称，在上面 web.xml 中的配置代码中即为 springmvc，而-servlet.xml 是配置文件名的固定写法，所以应用程序会在 WEB-INF 下寻找 springmvc-servlet.xml。

11.3.2 Controller 注解类型

org.springframework.stereotype.Controller 注解类型用于指示 Spring 类的实例是一个控制器，其注解形式为@Controller。该注解在使用时不需要再实现 Controller 接口，只需要将@Controller 注解加入控制器类上，然后通过 Spring 的扫描机制找到标注了该注解的控制器即可。

【示例 11-2】@Controller 注解在控制器类中的使用示例如下。

```
package com.ssm.controller;
import org.springframework.stereotype.Controller;
……
//Controller 注解
@Controller
public class ControllerTest{
    ……
}
```

为了保证 Spring 能够找到控制器类，还需要在 Spring MVC 的配置文件中添加相应的扫描配置信息，具体如下。

（1）在配置文件的声明中引入 spring-context。

（2）使用<context: component-scan>元素指定需要扫描的类包。

完整的配置文件如文件 11.5 所示。

文件 11.5 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!--指定需要扫描的包 -->
    <context:component-scan base-package="com.ssm.controller" />
</beans>
```

在文件 11.5 中，<context: component-scan>元素的属性 base-package 指定了需要扫描的类包为 com.ssm.controller。在运行时，该类包及其子包下所有标注了注解的类都会被 Spring 所处理。与实现了 Controller 接口的方式相比，使用注解的方式显然更加简单。同时，Controller 接口的实现类只能处理一个单一的请求动作，而基于注解的控制器可以同时处理多个请求动作，在使用上更加灵活。

因此，在实际开发中通常都会使用基于注解的形式。

注　意
使用注解方式时，程序的运行需要依赖 Spring 的 AOP 包，因此需要向 lib 目录中添加 spring.aop-4.3.6 RELEASE.jar，否则程序运行时会报错。

11.3.3　RequestMapping 注解类型

1. @RequestMapping 注解的使用

Spring 通过@Controller 注解找到相应的控制器类后，还需要知道控制器内部对每一个请求是如何处理的，这就需要使用 org.springframework.web.bind.annotation.RequestMapping 注解类型。RequestMapping 用于映射一个请求或一个方法，其注解形式为@RequestMapping，可以使用该注解标注在一个方法或一个类上。

（1）标注在方法上

当标注在一个方法上时，该方法将成为一个请求处理方法，它会在程序接收到对应的 URL 请求时被调用。

【示例 11-3】使用@RequestMapping 注解标注在方法上的示例如下。

```
package com.ssm.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
......
//Controller 注解
@Controller
public class AnnotationControllerTest {
    //@RequestMapping 注解标注在方法上
    @RequestMapping(value="/annotationController")
    public ModelAndView handleRequest(HttpServletRequest arg0,
                                HttpServletResponse arg1) throws Exception {
    ......
    return m;
    }
}
```

使用@RequestMapping 注解后，上述代码中的 handleRequest()方法就可以通过地址 http://localhost:8080/chapter11/annotationController 进行访问。

（2）标注在类上

当标注在一个类上时，该类中的所有方法都将映射为相对于类级别的请求，表示该控制器所处理的所有请求都被映射到 value 属性值所指定的路径下。

【示例 11-4】使用@RequestMapping 注解标注在类上的示例如下。

```
package com.ssm.controller;
import org.springframework.stereotype.Controller;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
......
//Controller 注解
@Controller
//@RequestMapping 注解标注在类上
@RequestMapping(value="/controll")
public class AnnotationControllerTest {
    @RequestMapping(value="/annotationController")
    public ModelAndView handleRequest(HttpServletRequest arg0,
                               HttpServletResponse arg1) throws Exception {
     ......
     return m;
     }
}
```

由于在类上添加了@RequestMapping 注解，并且其 value 属性值为“/controll”，因此上述代码方法的请求路径将变为 http://localhost:8080/chapter1 1/controll/annotationController。如果该类中还包含其他方法，那么在其他方法的请求路径中也需要加入“/ control”。

2 .@RequestMapping 注解的属性

@RequestMapping 注解除了可以指定 value 属性外，还可以指定一些其他属性，如表 11.1 所示。

表 11.1　@RequestMapping 注解的属性

属性名	类型	描述
name	String	可选属性，用于为映射地址指定别名
value	String[]	可选属性，同时也是默认属性，用于映射一个请求和一种方法，可以标注在一个方法或一个类上
method	RequestMethod[]	可选属性，用于指定该方法用于处理哪种类型的请求方式，其请求方式包括 GET、POST、HEAD、OPTIONS、PUT、PATCH、DELETE 和 TRACE，例如 method= RequestMethod. GET 表示只支持 GET 请求，如果需要支持多个请求方式，就需要通过“{}”写成数组的形式，并且多个请求方式之间是有英文逗号分隔的
params	String[]	可选属性，用于指定 Request 中必须包含某些参数的值，才可以通过其标注的方法处理
headers	String[]	可选属性，用于指定 Request 中必须包含某些指定的 header 的值，才可以通过其标注的方法处理
consumes	String[]	可选属性，用于指定处理请求的提交内容类型（Content-Type），比如 application/json、text/html 等
produces	String[]	可选属性，用于指定返回的内容类型，返回的内容类型必须是 request 请求头（Accept）中所包含的类型

在表 11.1 中，所有属性都是可选的，但其默认属性是 value。当 value 是其唯一属性时，可以省略属性名，例如下面两种标注的含义相同。

```
@RequestMapping(value="/annotationController")
@RequestMapping("/annotationController")
```

3. 组合注解

前面已经对@RequestMapping 注解及其属性进行了详细讲解，而在 Spring 4.3 版本中引入了组合注解来帮助简化常用的 HTTP 方法的映射，并更好地表达被注解方法的语义。其组合注解如下所示。

- @GetMapping：匹配 GET 方式的请求。
- @PostMapping：匹配 POST 方式的请求。
- @PutMapping：匹配 PUT 方式的请求。
- @DeleteMapping：匹配 DELETE 方式的请求。
- @PatchMapping：匹配 PATCH 方式的请求。

以@GetMapping 为例，该组合注解是@RequestMapping(method= RequestMethod.GET)的缩写，它会将 Http Get 映射到特定的处理方法上。在实际开发中，传统的@RequestMapping 注解使用方式如下。

```
@RequestMapping(value="/user/{id}",method=RequestMethod.GET)
public String selectUserById(string id){
     ……
}
```

而使用新注解@GetMapping 后，可以省略 method 属性，从而简化代码，使用方式如下。

```
@GetMapping(value="/user/{fid}")
public String selectUserById(string id){
     ……
}
```

4. 请求处理方法的参数类型和返回类型

在控制器类中，每一个请求处理方法都可以有多个不同类型的参数，以及一个多种类型的返回结果。例如在 11.2 节的入门案例中，handleRequest()方法的参数就是对应请求的 HttpServletRequest 和 HttpServletResponse 两种参数类型。除此之外，还可以使用其他的参数类型，例如在请求处理方法中需要访问 HttpSession 对象，就可以添加 HttpSession 作为参数，Spring 会将对象正确地传递给方法，其使用示例如下。

```
@RequestMapping(value="/annotationController ")
public ModelAndView(httpSession session)
{
    ……
    return m;
}
```

在请求处理方法中，可以出现的参数类型如下。

- javax.servlet.ServletRequest/javax.servlet.http.HttpServletRequest

- javax.servlet.ServletResponse/javax.servlet.http.HttpServletResponse
- javax.servlet.http.HttpSession
- org.springframework.web.context.request.WebRequest 或
- org.springframework.web.context.request.NativeWebRequest
- java.util.Locale
- java.util.TimeZone (Java 6+)/java.time.Zoneld(on Java 8)
- java.io.InputStream/java.io.Reader
- Java.io.OutputStream/java.io.Writer
- org.springframework.http.HttpMethod
- java.security.Principal
- @PathVariable、@MatrixVariable、@RequestParam、@RequestHeader、@RequestBody、
- @RequestPart、@SessionAttribute、@RequestAttribute 注解
- HttpEntity<?>
- java.util.Map/org.springframework.ui.Model/lorg.springframework.ui.ModelMap
- org.springframework.web.servlet.mvc.support.RedirectAttributes
- org.springframework.validation.Errors/org.springframework.validation.BindingResult
- org.springframework.web.bind.support.SessionStatus
- org.springframework.web.util.UriComponentsBuilder

注　意

org.springframework.ui.Model 类型不是一个 Servlet API 类型，而是一个包含 Map 对象的 Spring MVC 类型。如果方法中添加了 Model 参数，那么每次调用该请求处理方法时，Spring MVC 都会创建 Model 对象，并将其作为参数传递给方法。

在 11.2 节的入门案例中，请求处理方法返回的是一个 ModelAndView 类型的数据。除了这种类型外，请求处理方法还可以返回其他类型的数据。Spring MVC 所支持的常见方法返回类型如下。

- ModelAndView
- Model
- Map
- View
- String
- void
- HttpEntity<?>或 ResponseEntity<?>
- Callable<?>
- DeferredResult<?>

在上述列举的返回类型中，常见的返回类型有 ModelAndView、String 和 void。其中，ModelAndView 类型中可以添加 Model 数据，并指定视图；String 类型的返回值可以跳转视图，但不能携带数据；而 void 类型主要在异步请求时使用，它只返回数据，而不会跳转视图。

由于 ModelAndView 类型未能实现数据与视图之间的解耦，因此在开发时，方法的返回类型通常都会使用 String。既然 String 类型的返回值不能携带数据，那么在方法中是如何将数据带入视图页面呢？这就用到了上面所讲解的 Model 参数类型，通过该参数类型即可添加需要在视图中显示的属性。

返回 String 类型方法的示例代码如下。

```
@RequestMapping(value="/ annotationController")
public String handleRequest(HttpServletRequest arg0, HttpServletResponse arg1,
                                    Model model) throws Exception {
    model.addAttribute("msg","第一个 Spring MVC 程序");
    return "/WEB-INF/jsp/welcome.jsp";
}
```

在上述方法代码中增加了一个 Model 类型的参数，通过该参数实例的 addAttribute()方法即可添加所需数据。String 类型除了可以返回上述代码中的视图页面外，还可以进行复位向与请求转发，具体方式如下。

（1）redirect 复位向

例如，在修改用户信息操作后，将请求复位向到用户查询方法的实现代码如下。

```
@RequestMapping(value="/update")
public string update(httpServletRequest request,HttpServletRespoNse response,
                Model model){
    ……
    //复位向请求路径
    return "redirect: queryUser";
}
```

（2）forward 请求转发

例如，用户执行修改操作时，转发到用户修改页面的实现代码如下。

```
@RequestMapping(value="/toEdit")
public string toEdit(httpServletRequest request,HttpServletRespoNse response,
               Model model){
    ……
    //请求转发
    return "forward:editUser";
}
```

关于复位向和转发的具体使用，在本书后面章节中会有具体的应用案例。

11.3.4 ViewResolver（视图解析器）

Spring MVC 中的视图解析器负责解析视图，可以在配置文件中定义一个 ViewResolver 来配置视图解析器，其配置示例如下。

```
<!-- 定义视图解析器 -->
```

```
<bean id="viewResolver "
     class=" org.springframework.web.servlet.view.InternalResourceViewResolver ">
   <!-- 设置前缀 -->
   <property name="prefix" value="/WEB-INF/jsp/" />
   <!--设置后缀-->
   <property name="suffix" value=".isp" />
</bean>
```

在上述代码中定义了一个 id 为 viewResolver 的视图解析器，并设置了视图的前缀和后缀属性。这样设置后，方法中所定义的 view 路径将可以简化。例如，11.2 节的入门案例中的逻辑视图名只需设置为“welcome”，而不再需要设置为“/WEB-INF/jsp/welcome.jsp”，在访问时视图解析器会自动地增加前缀和后缀。

11.4　应用案例——基于注解的 Spring MVC 应用

为了帮助读者熟悉掌握 Spring MVC 的核心类和注解的使用，接下来将以注解的方式对入门案例进行改写，具体实现步骤如下。

11.4.1　搭建项目环境

在 Eclipse 中创建一个名为 chapter11_2 的 Web 项目，将 chapter11 项目中的所有 JAR 包以及编写的所有文件复制到 chapter11_2 项目，并向 lib 目录添加 Spring AOP 所需的 JAR（spring-aop-4.3.6.RELEASE.Jar）。

11.4.2　修改配置文件

在 springmvc-config.xml 中添加注解扫描配置，并定义视图解析器，如文件 11.6 所示。

文件 11.6　springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!--指定需要扫描的包 -->
    <context:component-scan base-package="com.ssm.controller" />
    <!-- 定义视图解析器 -->
    <bean id="viewResoler"
```

```
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- 设置前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!-- 设置后缀 -->
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

在文件 11.6 中，首先通过组件扫描器指定了需要扫描的包，然后定义了视图解析器，并在视图解析器中设置了视图文件的路径前缀和文件后缀名。

11.4.3 修改 Controller 类

修改 ControllerTest 类，在类和方法上添加相应注解，如文件 11.7 所示。

文件 11.7 ControllerTest.java

```
package com.ssm.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
//@Controller 注解
@Controller
//@RequestMapping 注解
@RequestMapping(value = "/controll")
public class ControllerTest {
    @RequestMapping(value = "/annotationController")
    public String handleRequest(HttpServletRequest arg0, HttpServletResponse arg1,
                   Model model) throws Exception {
     model.addAttribute("msg", "第一个 Spring MVC 程序");
     return "welcome";
    }
}
```

在文件 11.7 中使用@Controller 注解来标注控制器类，并使用@RequestMapping 注解标注在类名和方法名上来映射请求方法。在项目启动时，Spring 就会扫描到此类，以及此类中标注了@RequestMapping 注解的方法。由于标注在类上的@RequestMapping 注解的 value 值为“/controll”，因此类中所有请求方法的路径都需要加上“/controll”。由于类中的 handleRequest()方法的返回类型为 String，而 String 类型的返回值又无法携带数据，因此需要通过参数 Model 对象的 addAttribute()方法来添加数据信息。因为在配置文件的视图解析器中定义了视图文件的前缀和后缀名，所以 handleRequest()方法只需返回视图名“welcome”即可，在访问此方法时，系统会自动访问“WEB-INF/jsp/”路径下名称为 welcome 的 jsp 文件。

11.4.4　启动项目，测试应用

将项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter11_2/controll/annotationController，其显示效果如图 11.3 所示。从中可以看出，通过注解的方式同样实现了第一个 Spring MVC 程序的运行。

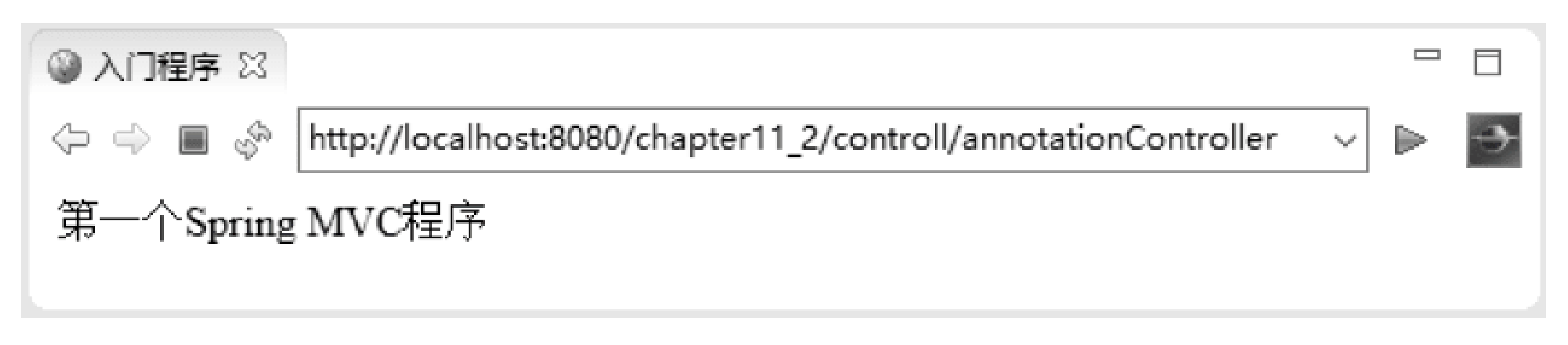


图 11.3　运行页面效果

11.5　习　　题

1. 请简述 Spring MVC 框架的执行流程，并参照 11.2 节的入门案例进行实践。
2. 对 11.2 节的入门案例通过注解的方式进行实践。

第 12 章

Spring MVC 数据绑定

在实际的开发中，客户端请求后台处理方法时可以传递参数，且参数包含多种类型。Spring MVC 框架是如何绑定并获取请求参数呢？本章将详细讲解。

本章主要涉及的知识点如下。

- Spring MVC 中的数据绑定的相关概念。
- Spring MVC 中的几种数据绑定类型。
- Spring MVC 数据绑定的使用。

12.1 数据绑定概述

在执行程序时，Spring MVC 根据客户端请求参数的不同将请求消息中的信息以一定的方式转换并绑定到控制器类的方法参数中。这种将请求消息数据与后台方法参数建立连接的过程就是 Spring MVC 中的数据绑定。

在数据绑定过程中，Spring MVC 框架会通过数据绑定组件（DataBinder）将请求参数串的内容进行类型转换，然后将转换后的值赋给控制器类中方法的形参，这样后台方法就可以正确绑定并获取客户端请求携带的参数。具体的信息处理过程的步骤如下。

步骤01 Spring MVC 将 ServletRequest 对象传递给 DataBinder。

步骤02 将处理方法的入参对象传递给 DataBinder。

步骤03 DataBinder 调用 ConversionService 组件进行数据类型转换、数据格式化等工作，并将 ServletRequest 对象中的消息填充到参数对象中。

步骤04 调用 Validator 组件对已经绑定了请求消息数据的参数对象进行数据合法性校验。

步骤05 校验完成后会生成数据绑定结果 BindingResult 对象，Spring MVC 会将 BindingResult 对象中的内容赋给处理方法的相应参数。

根据客户端请求参数类型和个数的不同，将 Spring MVC 中的数据绑定主要分为简单数据绑定和复杂数据绑定，接下来将讲解这两种类型的数据绑定。

12.2 简单数据绑定

简单数据绑定包括绑定默认数据类型、绑定简单数据类型、绑定 POJO 类型、绑定包装 POJO 等。

12.2.1 绑定默认数据类型

当前端请求的参数比较简单时，可以在后台方法的形参中直接使用 Spring MVC 提供的默认参数类型进行数据绑定。

常用的默认参数类型如下。

- HttpServletRequest：通过 request 对象获取请求信息。
- HttpServletResponse：通过 response 对象处理响应信息。
- HttpSession：通过 session 对象得到 session 中存储的对象。
- Model/ModelMap：Model 是一个接口，ModelMap 是一个接口实现，作用是将 Model 数据填充到 request 域。

【示例 12-1】下面以 HttpServletRequest 类型的使用为例进行演示说明，具体步骤如下。

步骤01 在 Eclipse 中创建一个名为 chapter12 的 Web 项目，然后将 Spring MVC 相关 JAR 包添加到项目的 lib 目录下，并发布到类路径中。添加 JAR 包后的目录如图 12.1 所示。

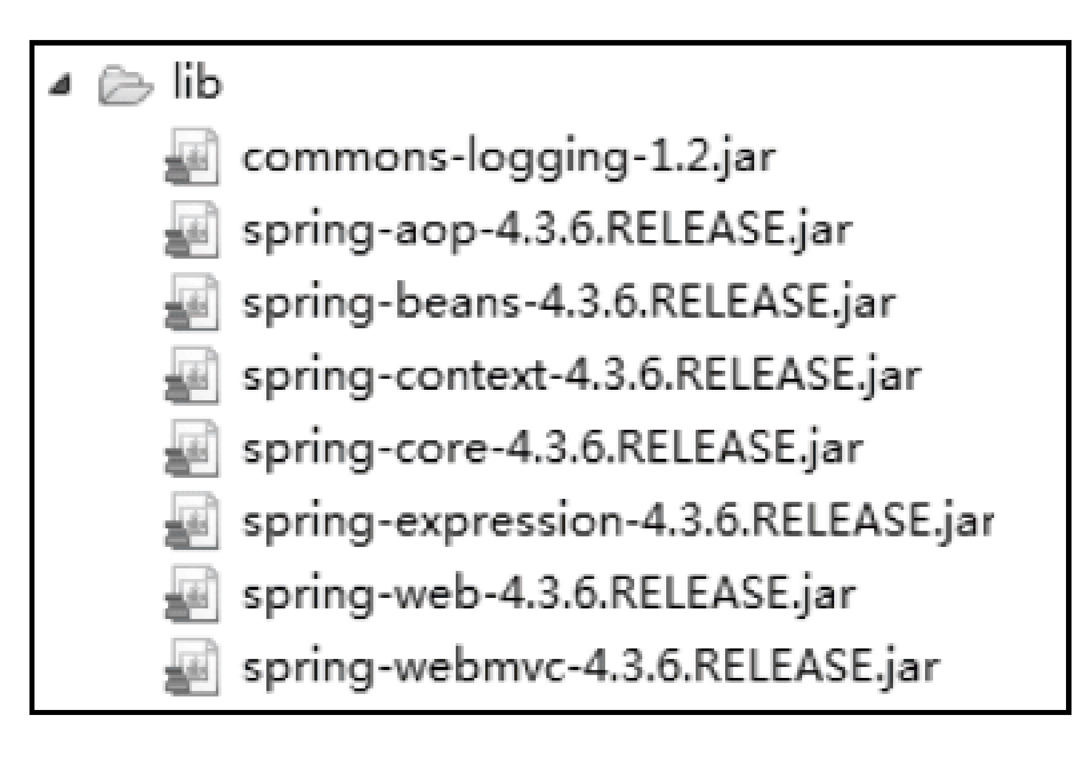


图 12.1 Spring MVC 相关 JAR 包

步骤02 在 web.xml 中配置 Spring MVC 的前端控制器等信息，如文件 12.1 所示。

文件 12.1 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
            xmlns="http://java.sun.com/xml/ns/javaee"
            xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         id="WebApp_ID" version="3.0">
   <servlet>
     <!-- 配置前端过滤器 -->
     <servlet-name>springmvc</servlet-name>
     <servlet-class>org.srpingframework.web.servlet.DispatcherServlet</servlet-class>
     <!-- 初始化时加载配置文件 -->
     <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>classpath:springmvc-config.xml</param-value>
     </init-param>
     <!-- 表示容器在启动时立即加载 Servlet -->
     <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
     <servlet-name>springmvc</servlet-name>
     <url-pattern>/</url-pattern>
      </servlet-mapping>
</web-app>
```

步骤03 在 src 目录下创建 Spring MVC 的核心配置文件 springmvc-config.xml，在该文件中配置组件扫描器和视图解析器，如文件 12.2 所示。

文件 12.2 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!--指定需要扫描的包 -->
    <context:component-scan base-package="com.ssm.controller" />
    <!-- 定义视图解析器 -->
    <bean id="viewResoler"
     class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     <!-- 设置前缀 -->
     <property name="prefix" value="/WEB-INF/jsp/" />
     <!-- 设置后缀 -->
     <property name="suffix" value=".jsp/" />
    </bean>
</beans>
```

步骤04 在 src 目录下创建一个 com.ssm.controller 包，在该包下创建一个用于用户操作的控

制器类 UserController，编写后的代码如文件 12.3 所示。

文件 12.3　UserController.java

```java
package com.ssm.controller;
import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
//@Controller 注解
@Controller
public class UserController {
//@RequestMapping 注解在方法上
    @RequestMapping(value="/selectUser")
    public String selectUser(HttpServletRequest request) {
     //获取请求地址中的参数 id 的值
     String id=request.getParameter("id");
     System.out.println("id="+id);
     return "success";
    }
}
```

在文件 12.3 中，使用注解方式定义了一个控制器类，同时定义了方法的访问路径。在方法参数中使用了 HttpServletRequest 类型，并通过该对象的 getParameter()方法获取了指定的参数。为了方便查看结果，将获取的参数进行输出打印，最后返回一个名为“success”的视图，Spring MVC 会通过视图解析器在“WEB-INF/jsp”路径下寻找“success.jsp”文件。

注　意

后台在编写控制器类时，通常会根据需要操作的业务对控制器类进行规范命名。例如要编写一个对用户操作的控制器类，可以将控制器类命名为 UserController，然后在该控制器类中就可以编写任何有关用户操作的方法。

步骤05 在 WEB-INF 目录下创建一个名为 jsp 的文件夹，然后在该文件夹中创建页面文件 success.jsp，该页面只作为正确执行操作后的响应页面，没有其他业务逻辑，如文件 12.4 所示。

文件 12.4　success.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                       "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>入门程序</title>
  </head>
  <body>
         ok! ,执行成功。
```

```
    </body>
</html>
```

步骤06 将 chapter12 项目发布到 Tomcat 服务器并启动，在浏览器中访问地址 http://localhost:8080/chapter12?selectUser?id=1，其显示效果如图 12.2 所示。

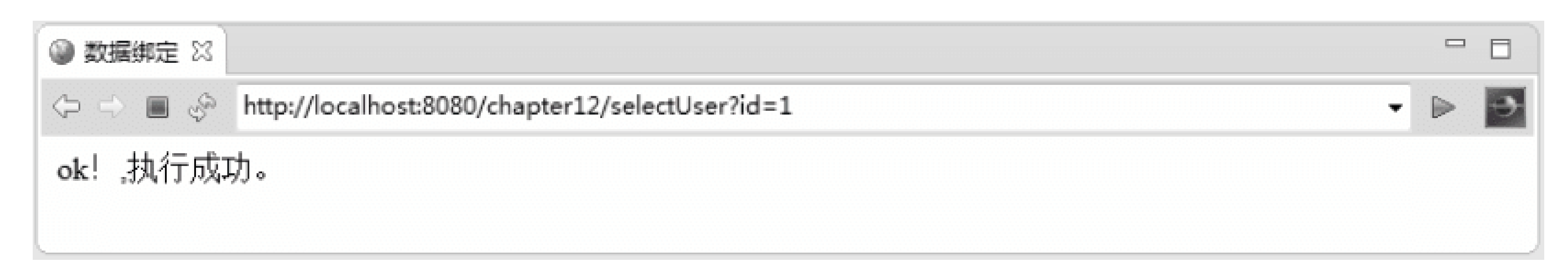


图 12.2 执行结果（success.jsp 页面）

此时的控制台输出信息如图 12.3 所示。从中可以看出，后台方法已从请求地址中正确地获取到了 id 的参数信息，这说明使用默认的 HttpServletRequest 参数类型已经完成了数据绑定。

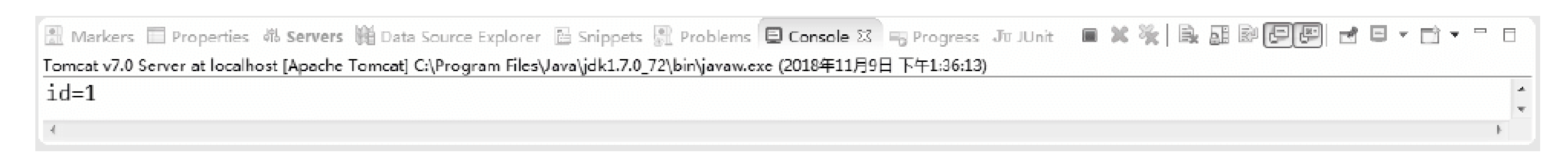


图 12.3 运行结果

12.2.2 绑定简单数据类型

简单数据类型的绑定就是指 Java 中几种基本数据类型的绑定，如 int、String、Double 等类型。

【示例 12-2】这里仍然以 12.2.1 小节中参数 id 为 1 的请求为例来讲解简单数据类型的绑定。

首先修改控制器类，将控制器类 UserController 中的 selectUser()方法的参数修改为使用简单数据类型的形式，修改后的代码如下。

```
@RequestMapping(value="/selectUser")
public String selectUser(Integer id) {
     System.out.println("id="+id);
     return "success";
}
```

与默认数据类型案例中的 selectUser()方法相比，此方法中只是将 HttpServletRequest 参数类型替换为了 Integer 类型。启动项目重新运行，会得到与图 12.2 和图 12.3 相同的运行结果，这说明使用简单的数据类型同样完成了数据绑定。

> **注　意**
>
> 有时前端请求中参数名和后台控制器类方法中的形参名不一样，就会导致后台无法正确绑定并接收到前端请求的参数。为此，Spring MVC 提供了@RequestParam 注解来进行间接数据绑定。

@RequestParam 注解主要用于定义请求中的参数，在使用时可以指定它的 4 个属性，如表 12.1 所示。

表 12.1　@RequestParam 注解的属性及说明

属性	说明
value	name 属性的别名，这里指参数的名字，即入参的请求参数名字，如 value="user_id" 表示请求的参数中名字为 user_id 的参数的值将传入。如果只使用 vaule 属性，就可以省略 value 属性名
name	指定请求头绑定的名称
required	用于指定参数是否必需，默认是 true，表示请求中一定要有相应的参数
defaultValue	默认值，表示如果请求中没有同名参数的默认值

@RequestParam 注解的使用非常简单，假设浏览器中的请求地址为 http://localhost:8080/chapter12/selectUser? User_id=1，那么在后台 selectUser()方法中的使用方式如下。

```
@RequestMapping(value="/selectUser")
public String selectUser(@RequestParam (value="user_id")Integer id) {
     System.out.println("id="+id);
     return "success";
}
```

上述代码会将请求中 user_id 参数的值 1 赋给方法中的 id 形参。这样通过输出语句就可以输出 id 形参中的值。

12.2.3　绑定 POJO 类型

在使用简单数据类型绑定时，可以很容易地根据具体需求来定义方法中的形参类型和个数，然而在实际应用中，客户端请求可能会传递多个不同类型的参数数据，如果还使用简单数据类型进行绑定，就需要手动编写多个不同类型的参数，这种操作显然比较烦琐。此时就可以使用 POJO 类型进行数据绑定。

POJO 类型的数据绑定就是将所有关联的请求参数封装在一个 POJO 中，然后在方法中直接使用该 POJO 作为形参来完成数据绑定。

【示例 12-3】接下来通过一个用户注册案例来演示 POJO 类型数据的绑定，具体实现步骤如下。

步骤01 在 src 目录下创建一个 com.ssm.po 包，在该包下创建一个 User 类来封装用户注册的信息参数，如文件 12.5 所示。

文件 12.5　User.java

```
package com.ssm.po;
//用户类 User
public class User {
    private Integer id;
    private String username;
    private String password;
    public Integer getId() {
```

```
        return id;
        }
        public void setId(Integer id) {
        this.id = id;
        }
        public String getUsername() {
        return username;
        }
        public void setUsername(String username) {
        this.username = username;
        }
        public String getPassword() {
        return password;
       }
        public void setPassword(String password) {
        this.password = password;
        }
}
```

步骤02 在控制器 UserController 类中编写向注册页面跳转和接收用户注册信息的方法，代码如下所示。

```
//向注册页面跳转
@RequestMapping("/toRegister")
public String toRegister() {
    return "register";
}
//接收用户注册信息
@RequestMapping("/registerUser")
public String registerUser(User user) {
    String username=user.getUserName();
    String password=user.getPassword();
    System.out.println("username="+username);
    System.out.println("password="+password);
    return "success";
}
```

步骤03 在 WEB-INF 目录下创建一个用户注册页面 register.jsp，在该界面中编写用户注册表单，表单需要以 POST 方式提交，并且在提交时会发送一条以“registerUser”结尾的请求消息，如文件 12.6 所示。

文件 12.6　register.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>注册</title>
</head>
<body>
      <form action="${pageContext.request.contextPath}/registerUser" method="post">
    用户名：<input type="text" name="username" /> <br />
        密  码：<input type="password" name="password" /> <br />
        <input type="submit" value="注册" />
      </form>
 </body>
</html>
```

注　意

在使用 POJO 类型数据绑定时，前端请求的参数名（本例中指 form 表单内各元素的 name 属性值）必须与要绑定的 POJO 类中的属性名一样，这样才会自动将请求数据绑定到 POJO 对象中，否则后台接收的参数值为 null。

步骤04 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter12/toRegister，就会跳转到用户注册页面 register.jsp，如图 12.4 所示。

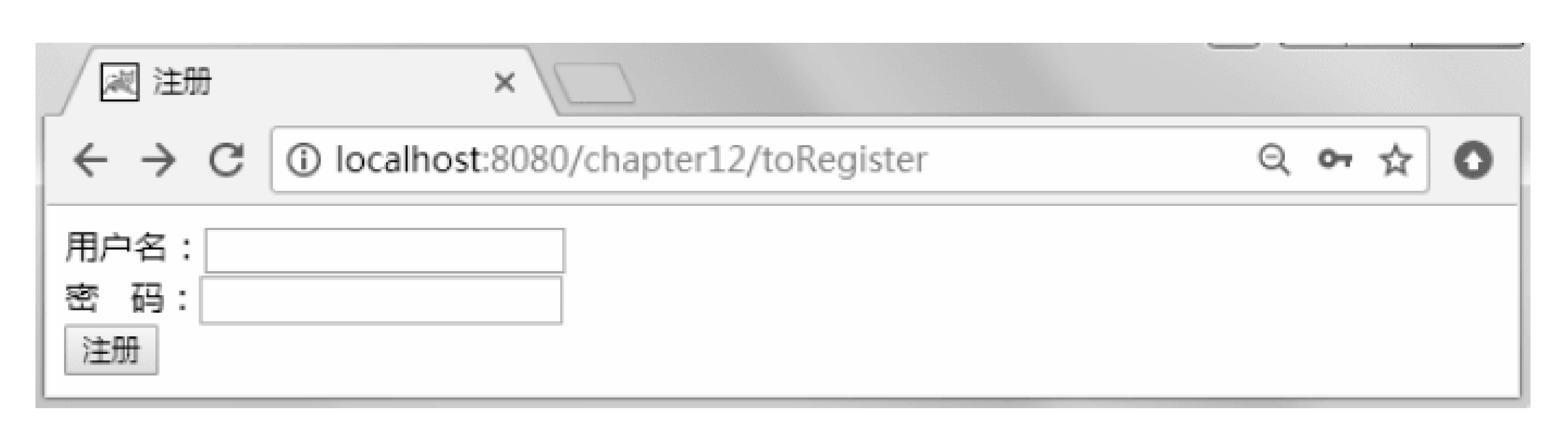


图 12.4　注册页面（register.jsp）

在图 12.4 中填写对应的用户名和密码，然后单击“注册”按钮即可完成模拟注册功能。这里假设用户注册的用户名和密码分别为“jack”和“jack_123”，当单击“注册”按钮后，浏览器会跳转到结果页面，此时控制台的输出结果如图 12.5 所示。

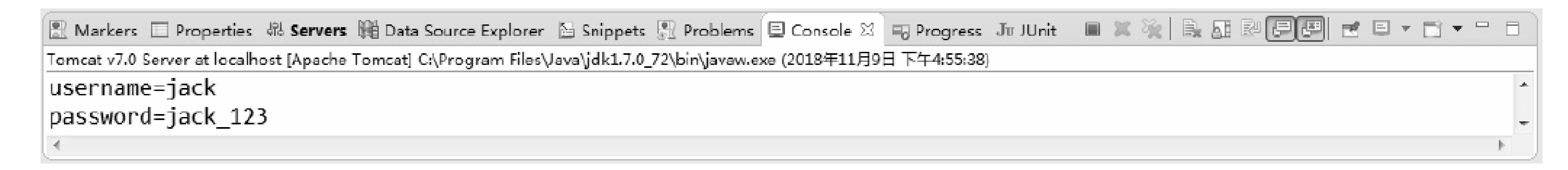


图 12.5　运行结果

从图 12.5 可以看出，使用 POJO 类型同样可以获取前端请求传递过来的数据信息，这就是 POJO 类型的数据绑定。

注 意

在前端请求中，难免会有中文信息传递，例如在图 12.4 的用户名和密码输入框中输入用户名“张三”和密码“123”时，虽然浏览器可以正确跳转到结果页面，但是在控制台中输出的中文信息会出现乱码。为了防止前端传入的中文数据出现乱码问题，可以使用 Spring 提供的编码过滤器来统一编码。要使用编码过滤器，只需要在 web.xml 中添加如下代码。

```
<!--配置编码过滤器-->
<filter>
        <filter-name>CharacterEncodingFilter</filter-name>

    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter
-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</filter>
```

上述代码中，通过<filter-mapping>元素的配置会拦截前端页面中的所有请求，并交由名称为 CharacterEncodingFilter 的编码过滤器类进行处理。在<filter>元素中，首先配置了编码过滤器类 org.springframework.web.filter.CharacterEncodingFilter，然后通过初始化参数设置统一的编码为 UTF-8。这样所有的请求信息都会以 UTF-8 的编码格式进行解析。

12.2.4 绑定包装 POJO

使用简单 POJO 类型已经可以完成多数的数据绑定，但有时客户端请求中传递的参数比较复杂。例如，在老师查询学生时，页面传递的参数可能包括班级名称和学生号等信息，这就包含班级和学生两个对象的信息。如果将班级和学生的所有查询条件都封装在一个简单 POJO 中，显然会比较混乱，这时就可以考虑使用包装 POJO 类型的数据绑定。

所谓包装 POJO，就是在一个 POJO 中包含另一个简单 POJO。例如，在学生对象中包含班级对象。这样在使用时，就可以通过学生查询到班级信息。

【示例 12-4】下面通过一个学生查询的案例来演示包装 POJO 数据绑定的使用，具体步骤如下。

步骤01 在 chapter12 项目的 com.ssm.po 包中创建班级类 Banji 和学生类 Student，Student 类用于封装学生和班级信息。两个类的代码如文件 12.7 和文件 12.8 所示。

文件 12.7　Banji.java

```
package com.ssm.po;
//班级类 Banji
public class Banji {
    private Integer banji_id;       //班级 id
    private String banji_name;      //班级名
    //其他的属性省略
    public Integer getBanji_id() {
     return banji_id;
    }
    public void setBanji_id(Integer banji_id) {
     this.banji_id = banji_id;
    }
    public String getBanji_name() {
     return banji_name;
    }
    public void setBanji_name(String banji_name) {
     this.banji_name = banji_name;
    }
}
```

文件 12.8　Student.java

```
package com.ssm.po;
//学生类
public class Student {
    private Integer stu_id;         //学生 id
    private String stu_name;        //学生姓名
    private Banji banji;            //学生所在班级
    //其他属性省略
    public Integer getStu_id() {
     return stu_id;
    }
    public void setStu_id(Integer stu_id) {
     this.stu_id = stu_id;
    }
    public String getStu_name() {
     return stu_name;
    }
    public void setStu_name(String stu_name) {
     this.stu_name = stu_name;
    }
    public Banji getBanji() {
     return banji;
    }
    public void setBanji(Banji banji) {
     this.banji = banji;
```

```
    }
}
```

在上述包装 POJO 类 Student 中，不仅可以封装学生的基本属性参数，还可以封装 Banji 类型的属性参数。

步骤02 在 com.ssm.controller 包中创建一个学生控制器类 StudentController，在该类中编写一个跳转到学生查询页面的方法和一个查询学生及班级信息的方法，如文件 12.9 所示。

文件 12.9　StudentController.java

```
package com.ssm.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.ssm.po.Banji;
import com.ssm.po.Student;
@Controller
public class StudentController {
    //向学生查询页面跳转
    @RequestMapping("/tofindStudentWithBanji")
    public String tofindStudentWithBanji(){
     return "student";
    }
    //查询学生和班级信息
    @RequestMapping("/findStudentWithBanji")
public String findStudentWithBanji(Student student){
     Integer stu_id=student.getStu_id();
     Banji banji=student.getBanji();
     String banji_name=banji.getBanji_name();
     System.out.println("stu_id="+stu_id);
     System.out.println("banji_name="+banji_name);
     return "success";
    }
}
```

在文件 12.9 中，通过访问页面跳转方法即可跳转到 student.jsp，而通过查询学生和班级信息方法，即可通过传递的参数条件调用 Service 中的相应方法来查询数据。这里只是为了讲解 POJO 的使用，所以只需要将传递过来的参数进行输出。

步骤03 在 WEB-INF/jsp 目录下创建一个班级学生查询页面 student.jsp，在页面中编写通过学生编号和所属班级作为查询条件来查询学生信息的代码，如文件 12.10 所示。

文件 12.10　student.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>学生查询</title>
</head>
<body>
    <form              action="${pageContext.request.contextPath}/findStudentWithBanji"
    method="post">
      学生编号：<input type="text" name="stu_id" /> <br />
      所属班级：<input type="text" name="banji.banji_name" /> <br />
      <input type="submit" value="查询" />
     </form>
 </body>
</html>
```

注　意

在使用包装 POJO 类型数据绑定时，前端请求的参数名编写必须符合以下两种情况。

①如果查询条件参数是包装类的直接基本属性，参数名就直接用对应的属性名，如上述代码中的 stu_id。

②如果查询条件参数是包装类中 POJO 的子属性，参数名就必须为【对象.属性】，其中【对象】要和包装 POJO 中的对象属性名称一致，【属性】要和包装 POJO 中的对象子属性一致，如上述代码中的 banji.banji_name。

步骤04 发布并启动项目，访问地址 http://localhost:8080/chapter12/tofindStudentWithBanji，显示效果如图 12.6 所示。

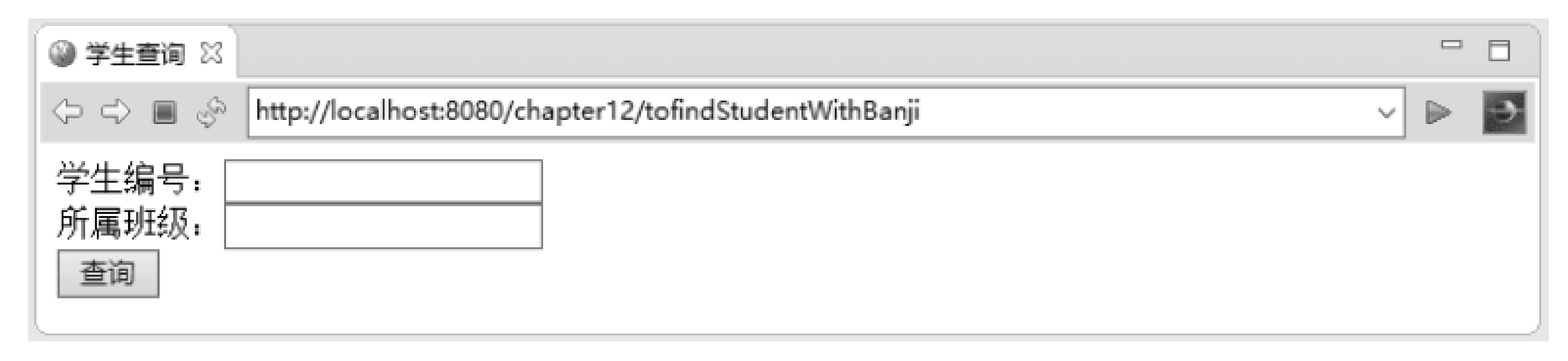


图 12.6　student.jsp 页面

在图 12.6 中填写学生编号为“101011”，所属班级为“软件工程”，单击“查询”按钮后，浏览器会跳转到 success.jsp 页面，此时控制台中的打印信息如图 12.7 所示。从图 12.7 中可以看出，使用包装 POJO 同样完成了数据绑定。



图 12.7　运行结果

除了上述几种简单数据绑定外，有些特殊类型的参数无法在后台进行直接转换，例如日期数据需要开发者自定义转换器（Converter）或格式化（Formatter）进行数据绑定。相关内容请读者查阅相关资料进行自学。

12.3 复杂数据绑定

实际项目开发中，除了简单数据类型外，还会经常遇到一些比较复杂的数据绑定问题，比如数组的绑定、集合的绑定，接下来将具体讲解一下数组绑定和集合绑定的使用。

12.3.1 绑定数组

在实际开发时，可能会遇到前端请求需要传递到后台一个或多个相同名称参数的情况（如批量删除），此时不适合采用简单数据绑定，而可以使用绑定数组的方式。

【示例 12-5】下面通过一个批量删除用户的例子来详细讲解绑定数组的操作。

步骤01 在 chapter12 项目的 WEB-INF/jsp 目录下创建一个展示课程信息的列表页面 course.jsp，代码如文件 12.11 所示。

文件 12.11 course.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>课程列表</title>
    </head>
    <body>
     <form action="${pageContext.request.contextPath}/deleteCourse" method="post">
         <table>
             <tr>
                 <td>选择</td>
                 <td>课程名</td>
             </tr>
             <tr>
                 <td><input name="ids" value="1" type="checkbox"></td>
                 <td>JAVA 程序设计</td>
             </tr>
             <tr>
                 <td><input name="ids" value="2" type="checkbox"></td>
                 <td>MySQL 数据库</td>
             </tr>
             <tr>
                 <td><input name="ids" value="3" type="checkbox"></td>
                 <td>JavaEE 应用开发</td>
             </tr>
```

```
            </table>
            <input type="submit" value="删除"/>
        </form>
      </body>
</html>
```

在上述页面代码中定义了 3 个 name 属性相同而 value 属性值不同的复选框控件，并在每一个复选框对应的行中编写了一个对应课程。在单击“删除”按钮执行删除操作时，表单会提交到一个以“/ deleteCourse”结尾的请求中。

步骤02 在控制器类 CourseController 中编写接收批量删除课程的方法（同时为了方便向课程列表页面跳转，还需增加一个向 course.jsp 页面跳转的方法），其代码如文件 12.12 所示。

文件 12.12　CourseController.java

```
package com.ssm.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class CourseController {
    //向课程页面跳转
    @RequestMapping("/toCourse")
    public String toCourse(){
        return "course";
    }
    //删除课程
    @RequestMapping("/deleteCourse")
    public String deleteCourse(Integer[] ids){
        if(ids!=null){
            //使用输出语句模拟已经删除的课程
            for(Integer id:ids){
                System.out.println("删除了 id 为"+id+"的课程");
            }
        }else{
            System.out.println("ids=null");
        }
        return "success";
    }
}
```

在上述代码中，先定义了一个向课程列表页面 course.jsp 跳转的方法 toCourse()，然后定义了一个接收前端批量删除用户的方法。在删除方法中使用了 Integer 类型的数组进行数据绑定，并通过 for 执行具体数据的删除操作。

步骤03 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter12/toCourse，显示效果如图 12.8 所示。

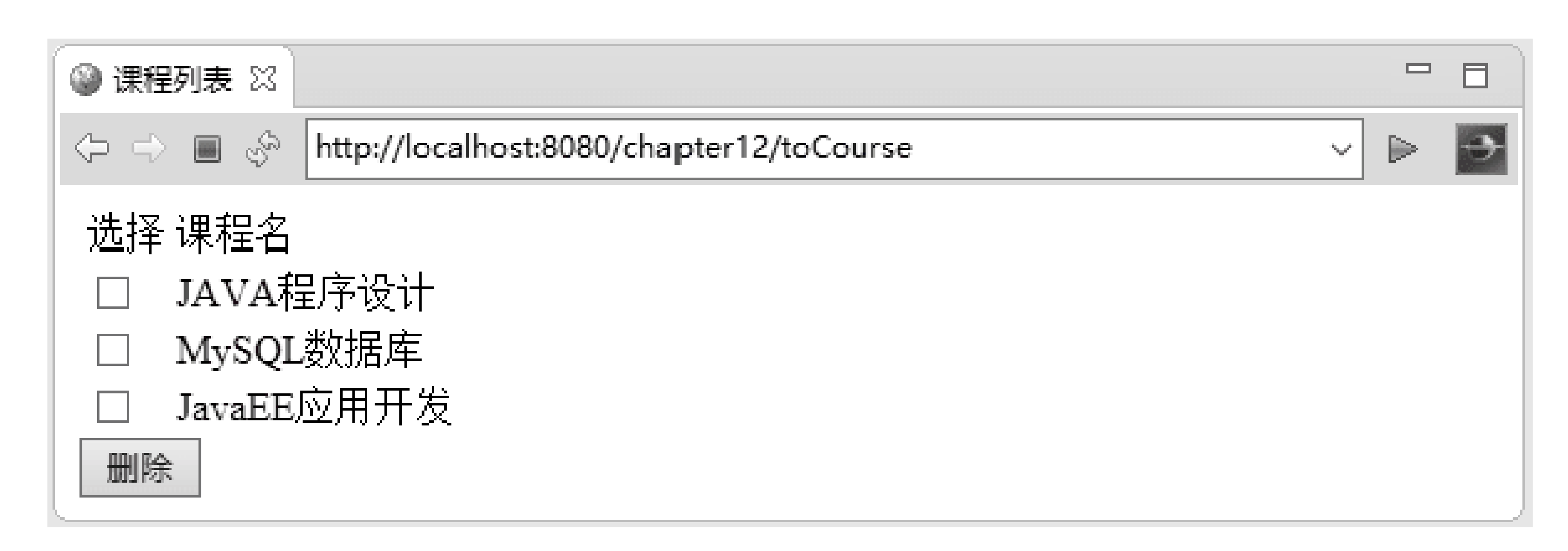


图 12.8　运行结果 course.jsp 课程列表页面

勾选图 12.8 中的全部复选框，然后单击“删除”按钮，程序在正确执行后会跳转到 success.jsp 页面。此时控制台的打印信息如图 12.9 所示。从图 12.9 中可以看出，已成功执行了批量删除操作，这说明已成功实现了数组类型的数据绑定。

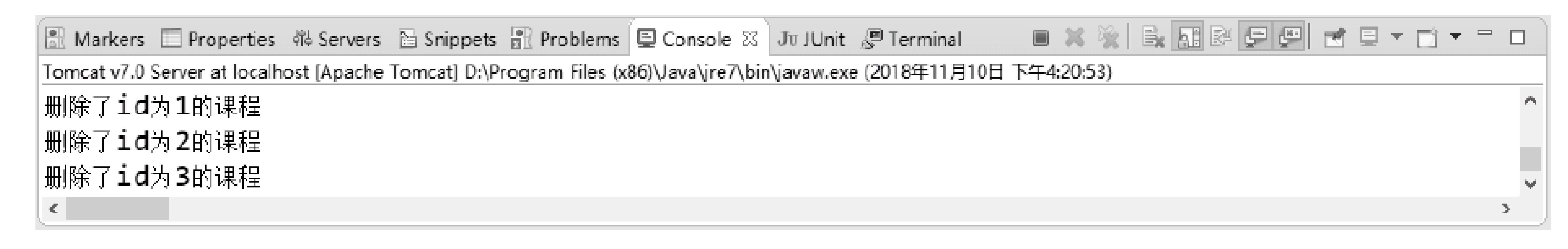


图 12.9　运行结果

12.3.2　绑定集合

在项目中，前端请求传递过来的数据可能会批量包含各种类型的数据，如 Integer、String 等。这种情况使用数组绑定是无法实现的。针对这种情况，可以使用集合数据绑定，即在包装类中定义一个包含对象类的集合，然后在接收方法中将参数类型定义为该包装类的集合。

【示例 12-6】下面以批量修改用户为例讲解一下集合数据绑定的使用。

步骤01 在 src 目录下创建一个 com.ssm.vo 包，并在包中创建包装类 UserVo 来封装课程集合属性，代码如文件 12.13 所示。

文件 12.13　UserVo.java

```
package com.ssm.vo;
import java.util.List;
import com.ssm.po.User;
/*
 * 用户包装类
 */
public class UserVo {
    private List<User> users;   //用户列表
    public List<User> getUsers() {
     return users;
    }
    public void setUsers(List<User> users) {
     this.users = users;
    }
}
```

步骤02 在控制器类 UserController 中编写接收批量修改用户的方法，以及向用户修改页面跳转的方法，其代码如下所示。

```
//向用户批量修改页面跳转
@RequestMapping("/toUserEdit")
public String toUserEdit() {
    return "user_edit";
}
//接收批量修改用户的方法
@RequestMapping("/editUsers")
public String editUsers(UserVo userList){
    //将所有用户数据封装到集合中
    List<User> users=userList.getUsers();
    for(User user:users){
        if(user.getId()!=null){
            System.out.println("删除了 id 为"+user.getId()+"的用户名为"+
                                                    user.getUsername());
        }
    }
    return "success";
}
```

在上述代码的两个方法中，通过 toUserEdit()方法将跳转到 user_edit.jsp 页面，通过 editUsers()方法将执行用户批量更新操作，其中该方法的 UserVo 类型参数用于绑定并获取页面传递过来的用户数据。

注　意
在使用集合数据绑定时，后台方法中不支持直接使用集合形参进行数据绑定，所以需要使用包装 POJO 作为参数，然后在包装 POJO 中包装一个集合属性。

步骤03 在项目的/WEB-INF/jsp 目录下创建页面文件 user_edit.jsp，如文件 12.14 所示。

文件 12.14　user_edit.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>修改用户</title>
    </head>
    <body>
     <form action="${pageContext.request.contextPath}/editUsers" method="post">
         <table>
             <tr>
```

```
                    <td>选择</td>
                    <td>用户名</td>
                </tr>
                <tr>
                    <td><input name="users[0].id" value="1" type="checkbox"></td>
                    <td>
                  <input name="users[0].username" value="zhangsan" type="checkbox">
                 </td>
                </tr>
                    <tr>
                    <td><input name="users[1].id" value="2" type="checkbox"></td>
                    <td>
                    <input name="users[1].username" value="lisi" type="checkbox">
                  </td>
                </tr>
            </table>
            <input type="submit" value="修改"/>
        </form>
    </body>
</html>
```

在上述页面代码中，模拟展示了 id 为 1、用户名为 zhangsan 和 id 为 2、用户名为 lisi 的两个用户。当单击“修改”按钮后，会将表单提交到一个以“editUsers”结尾的请求中。

步骤04 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter12/toUserEdit，显示效果如图 12.10 所示。

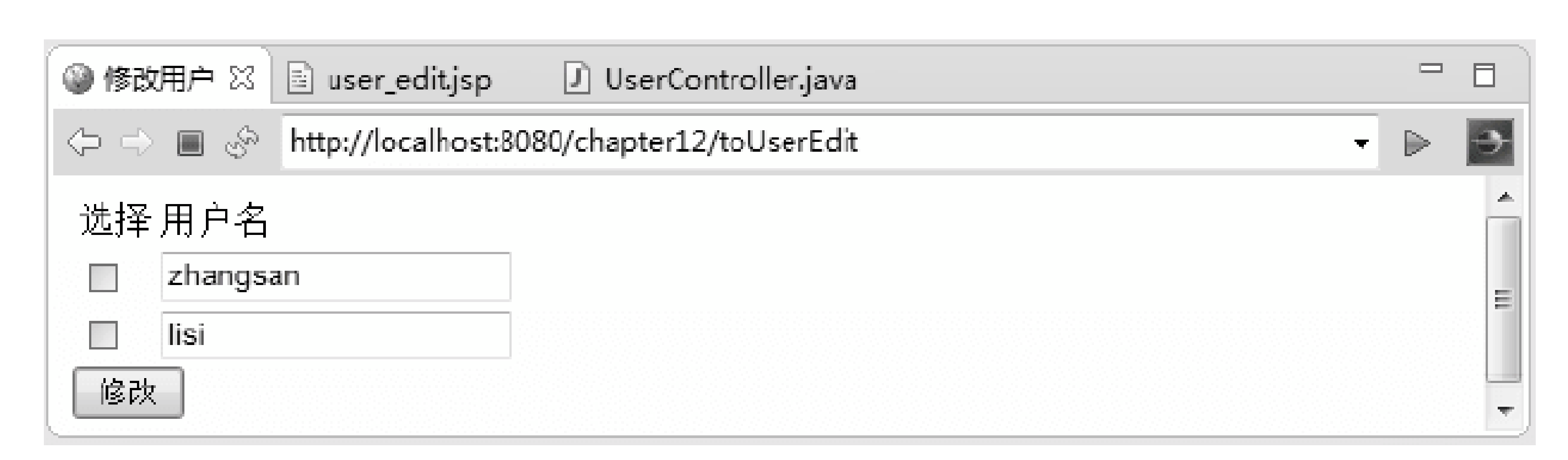


图 12.10　user_edit.jsp 页面

将图 12.10 中的用户名 zhangsan 改为 tom，lisi 改为 rose，并勾选两个数据前面的复选框，然后单击“修改”按钮后，浏览器会跳转到 success.jsp 页面中。此时控制台的打印信息如图 12.11 所示。从中可以看出，已经成功输出了请求中批量修改的用户信息，这就是集合类型的数据绑定。

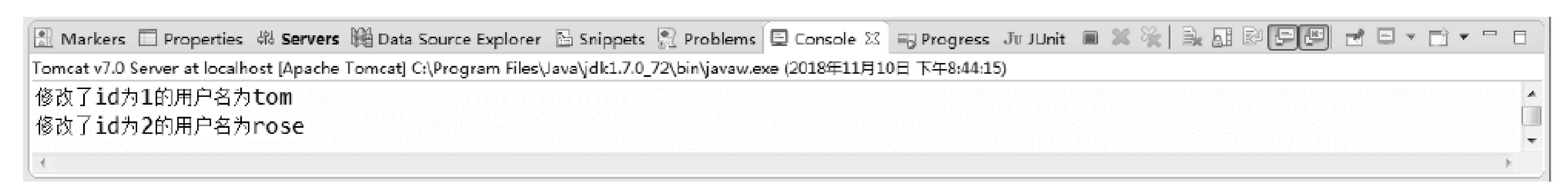


图 12.11　运行结果

12.4　习　　题

1. 请简述 Spring MVC 数据绑定的过程。
2. 请简述包装 POJO 类型绑定时的注意事项。

第 13 章

JSON 数据交互和 RESTful 支持

Spring MVC 中的数据绑定需要对传递数据的格式和类型进行转换，包括第 12 章讲解的简单类型数据和复杂类型数据，实际上还包括 JSON 等其他类型的数据。本章将讲解针对 Spring MVC 中 JSON 类型的数据交互和 RESTful 支持。

本章主要涉及的知识点如下。

- Spring MVC 中 JSON 数据交互的使用。
- RESTfU 风格的请求样式。
- Spring MVC 中 RESTful 风格请求的使用。

13.1 JSON 数据交互

JSON 与 XML 非常相似，都是用于存储数据的，但 JSON 相对于 XML 来说，解析速度更快，占用空间更小。

13.1.1 JSON 概述

JSON（JavaScript Object Notation，JS 对象标记）是一种轻量级的数据交换格式，最近几年才流行起来。JSON 是基于 JavaScript 的一个子集，使用了 C、C++、C#、Java、 JavaScript、Per、Python 等其他语言的约定，采用完全独立于编程语言的文本格式来存储和表示数据。这些特性使 JSON 成为理想的数据交互语言，它易于阅读和编写，同时也易于机器解析和生成。

与 XML 一样，JSON 也是基于纯文本的数据格式。初学者可以使用 JSON 传输一个简单的 String、Number、Boolean，也可以传输一个数组或者一个复杂的 Object 对象。JSON 有两种数据结构：对象

结构和数组结构。

1. 对象结构

对象结构以“{”开始，以“}”结束。中间部分由 0 个或多个以英文“,”分隔的“key:value”对构成，其中 key 和 value 之间也是以英文“:”间隔的。

对象结构的语法结构代码如下。

```
{
 key1: value1,
 key2: value2,
 ……
}
```

其中关键字（key）必须为 String 类型，值（value）可以是 String、Number、Object、Array 等数据类型。例如，一个 address 对象包含城市、街道等信息，使用 JSON 的表示形式如下。

```
{"city":"Guangzhou","street":"Zhongsan Road" }
```

2. 数组结构

数组结构以“[”开始，以“]”结束。中间部分由 0 个或多个以英文“,”分隔的值的列表组成。

数组结构的语法结构代码如下。

```
{
 value1,
 value2,
 ……
}
```

例如，一个数组包含 String、Number、Boolean、null 类型数据，使用 JSON 的表示形式如下。

```
{"jack",27,false,null}
```

JSON 这两种数据结构可以分别组合构成更为复杂的数据结构。例如，一个 person 对象包含 name、hobby 和 address 对象，其代码表现形式如下。

```
{
  "name":"zhangsan",
  "hobby":["羽毛球","读书"],
  "address":{
     "city": "Guangzhou",
     "street": "Zhongsan Road"
     }
}
```

注　意

如果使用 JSON 存储单个数据（如“abc”），一定要使用数组的形式，不要使用 Object 形式，因为 Object 形式必须是“名称:值”的形式。

13.1.2 JSON 数据转换

为了实现浏览器与控制器类（Controller）之间的数据交互，Spring 提供了一个 HttpMessageConverter<T>接口来完成此项工作。该接口主要用于将请求信息中的数据转换为一个类型为 T 的对象，并将类型为 T 的对象绑定到请求方法的参数中，或者将对象转换为响应信息传递给浏览器显示。

Spring 为 HttpMessageConverter<T>接口提供了很多实现类，这些实现类可以对不同类型的数据进行信息转换。其中 MappingJacksona2HttpMessageConverter 是 Spring MVC 默认处理 JSON 格式请求响应的实现类。该实现类利用 Jackson 开源包读写 JSON 数据，将 Java 对象转换为 JSON 对象和 XML 文档，同时可以将 JSON 对象和 XML 文档转换为 Java 对象。

要使用 MappingJacksona2HttpMessageConverter 对数据进行转换，就需要使用 Jackson 的开源包，开发时所需的开源包及其描述如下所示。

- jackson-annoations-2.8.8.jar：JSON 转换注解包。
- jackson-core-2.8.8.jar：JSON 转换核心包。
- Jackson- databind-2.8.8.jar：JSON 转换的数据绑定包。

以上 3 个 Jackson 的开源包可以通过链接“https://mvnrepository.com/artifact/com.fasterxml.jackson.core”下载得到。读者也可以下载更高版本的 JAR 包。

在使用注解式开发时，需要用到两个重要的 JSON 格式转换注解@RequestBody 和@ResponseBody。关于这两个注解的说明如表 13.1 所示。

表 13.1 JSON 数据交互注解及说明

注解	说明
@RequestBody	用于将请求体中的数据绑定到方法的形参中，该注解用在方法的形参上
@ResponseBody	用于直接返回 return 对象，该注解用在方法上

【示例 13-1】了解了 Spring MVC 中 JSON 数据交互需要使用的类和注解后，接下来通过一个案例来演示如何进行 JSON 数据的交互。

步骤01 创建项目并导入相关 JAR 包。使用 Eclipse 创建一个名为 chapter13 的 Web 项目，然后将 SpringMVC 相关 JAR 包、JSON 转换包添加到项目的 lib 目录中，并发布到类路径下。添加后的 lib 目录如图 13.1 所示。



图 13.1 项目相关的 JAR 包

步骤02 在 web.xml 中，对 Spring MVC 的前端控制器等信息进行配置，如文件 13.1 所示。

文件 13.1　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
                   id="WebApp_ID" version="3.0">
    <display-name>chapter13</display-name>
    <welcome-file-list>
     <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <!-- 配置前端控制器 -->
    <servlet>
     <!-- 配置前端过滤器 -->
     <servlet-name>springmvc</servlet-name>
     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
     <!-- 初始化时加载配置文件 -->
     <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>classpath:springmvc-config.xml</param-value>
     </init-param>
     <!-- 表示容器在启动时立即加载 Servlet -->
     <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
     <servlet-name>springmvc</servlet-name>
     <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

步骤03 在 src 目录下创建 Spring MVC 的核心配置文件 springmvc-config.xml，如文件 13.2 所示。

文件 13.2　springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
```

```
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!--指定需要扫描的包 -->
    <context:component-scan base-package="com.ssm.controller" />
    <!-- 配置注解驱动 -->
    <mvc:annotation-driven />
    <!-- 配置静态资源的访问映射，此配置中的文件将不被前端控制器拦截 -->
    <mvc:resources location="/js/" mapping="/js/**"></mvc:resources>
    <!-- 定义视图解析器 -->
    <bean id="viewResoler"
     class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     <!-- 设置前缀 -->
     <property name="prefix" value="/WEB-INF/jsp/" />
     <!-- 设置后缀 -->
     <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

在文件 13.2 中，不仅配置了组件扫描器和视图解析器，还配置了 Spring MVC 的注解驱动<mvc:annotation-driven/>和静态资源访问映射<mvc:resources…/>。其中<mvc:annotation-driven/>配置会自动注册 RequestMappingHandlerMapping 和 RequestMappingHandlerAdapter 两个 Bean，并提供对读写 XML 和读写 JSON 等功能的支持。<mvc:resources…/>元素用于配置静态资源的访问路径。由于在 web.xml 中配置的“/”会对页面中引入的静态文件进行拦截，而拦截后页面中将找不到这些静态资源文件，因此会引起页面报错。而增加了静态资源的访问映射配置后，程序会自动地去配置路径下找静态的内容。

<mvc:resources…/>中有两个重要属性 location 和 mapping。关于这两个属性的说明如表 13.2 所示。

表 13.2 <mvc:resources>标签属性及说明

属性	说明
location	用于定位需要访问的本地静态资源文件路径，具体到某个文件夹
mapping	匹配静态资源全路径，其中“/”表示文件夹及其子文件夹下的某个具体文件

步骤04 在 src 目录下创建一个 com.ssm.po 包，并在包中创建一个客户 Customer 类，该类用于封装 User 类型的请求参数，如文件 13.3 所示。

文件 13.3 Customer.java

```
package com.ssm.po;
//客户类 Customer
public class Customer {
    private Integer id;
    private String loginname;       //客户登录名
    private String nickname;    //昵称
    private String password;    //密码
    public Integer getId() {
```

```
        return id;
       }
      public void setId(Integer id) {
        this.id = id;
       }
      public String getLoginname() {
        return loginname;
       }
      public void setLoginname(String loginname) {
        this.loginname = loginname;
       }
      public String getNickname() {
        return nickname;
       }
      public void setNickname(String nickname) {
        this.nickname = nickname;
       }
      public String getPassword() {
        return password;
       }
      public void setPassword(String password) {
        this.password = password;
       }
      public String toString() {
        return "Customer [id=" + id + ", loginname=" + loginname + ",nickname=" + nickname +
                      ", password=" + password    + "]";
       }
  }
```

步骤05 在 WebContent 目录下创建页面文件 json.jsp 来测试 JSON 数据交互，如文件 13.4 所示。

文件 13.4　json.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
   <title>测试 JSON 交互</title>
   <script type="text/javascript"
    src="${pageContext.request.contextPath}/js/jquery-1.11.3.min.js">
   </script>
   <script type="text/javascript">
   function testJson(){
    //获取输入的客户信息
```

```
        var loginname=$("#loginname").val();
        var password=$("#password").val();
        $.ajax({
            url:"${pageContext.request.contextPath}/testJson",
            type:"post",
            //data 表示发送的数据
            data:JSON.stringfy({loginname:loginname,password:password}),
            // 定义发送请求的数据格式为 JSON 字符串
            contentType:"application/json;charset=UTF-8",
            //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
            dataType:"json",
            //成功响应的结果
            success:function(data){
                if(data!=null){
                    alert("您输入的登录名为："+data.loginname+"密码为："+data.password);
                }
            }
        });
        }
        </script>
        </head>
        <body>
        <form>
        登录名：<input type="text" name="loginname" id="loginname" /> <br />
        密码：<input type="password" name="password" id="password" /> <br />
        <input type="button" value="测试 JSON 交互" onclick="testJson()" />
        </form>
        </body>
</html>
```

在文件 13.4 中编写了一个测试 JSON 交互的表单，当单击“测试 JSON 交互”按钮时，会执行页面中的 testJson()函数。在函数中使用了 jQuery 的 AJAX 方式将 JSON 格式的登录名和密码传递到以“/ testJson”结尾的请求中。

在 AJAX 中包含 3 个特别重要的属性，其说明如下。

- data：请求时携带的数据，当使用 JSON 格式时，要注意编写规范。
- contentType：当请求数据为 JSON 格式时，值必须为 application/json。
- dataType：当响应数据为 JSON 时，可以定义 dataType 属性，并且值必须为 json。其中 dataType:"json" 可以省略不写，页面会自动识别响应的数据格式。

提　示
json.jsp 还需要引入 jquery.js 文件，本例中引入了 WebContent 目录下 js 文件夹中的 jquery-1.11.3.min.js。

步骤06 在 src 目录下创建一个 com.ssm.controller 包，在该包下创建一个用于客户操作的控

制器类 CustomerController，代码如文件 13.5 所示。

文件 13.5　CustomerController.java

```
package com.ssm.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import com.ssm.po.Customer;
@Controller
public class CustomerController {
    /*
     * 接收页面请求的 JSON 数据，并返回 JSON 格式结果
     */
    @ResponseBody
    public Customer testJson(@RequestBody Customer customer){
     //打印接收到的 JSON 格式数据
     System.out.println(customer);
     return customer;
    }
}
```

在文件 13.5 中，使用注解方式定义了一个控制器类，并编写了接收和响应 JSON 格式数据的 testJson()方法，在方法中接收并打印了接收到的 JSON 格式的用户数据，然后返回了 JSON 格式的用户对象。

方法中的@RequestBody 注解用于将前端请求体中的 JSON 格式数据绑定到形参 customer 上，@ResponseBody 注解用于直接返回 Custome 对象（当返回 POJO 对象时，会默认转换为 JSON 格式数据进行响应）。

步骤07 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter13/json.jsp，显示效果如图 13.2 所示。

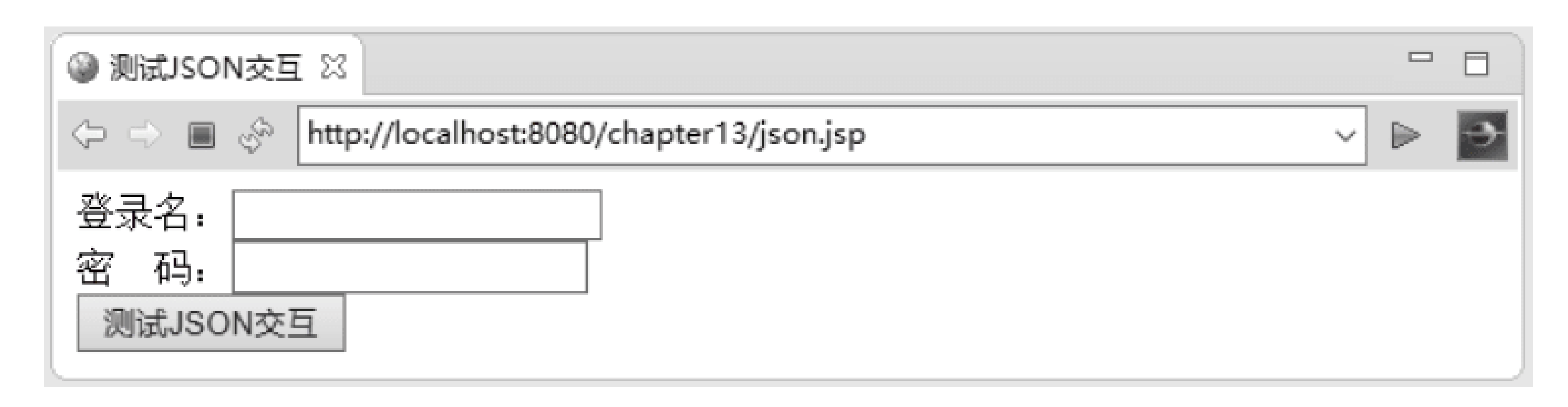


图 13.2　json.jsp 页面

在两个输入框中分别输入用户名“wujit”和密码“123456”后，单击“测试 JSON 交互”按钮，当程序正确执行时，页面中会弹出显示用户名和密码的弹出框，如图 13.3 所示。

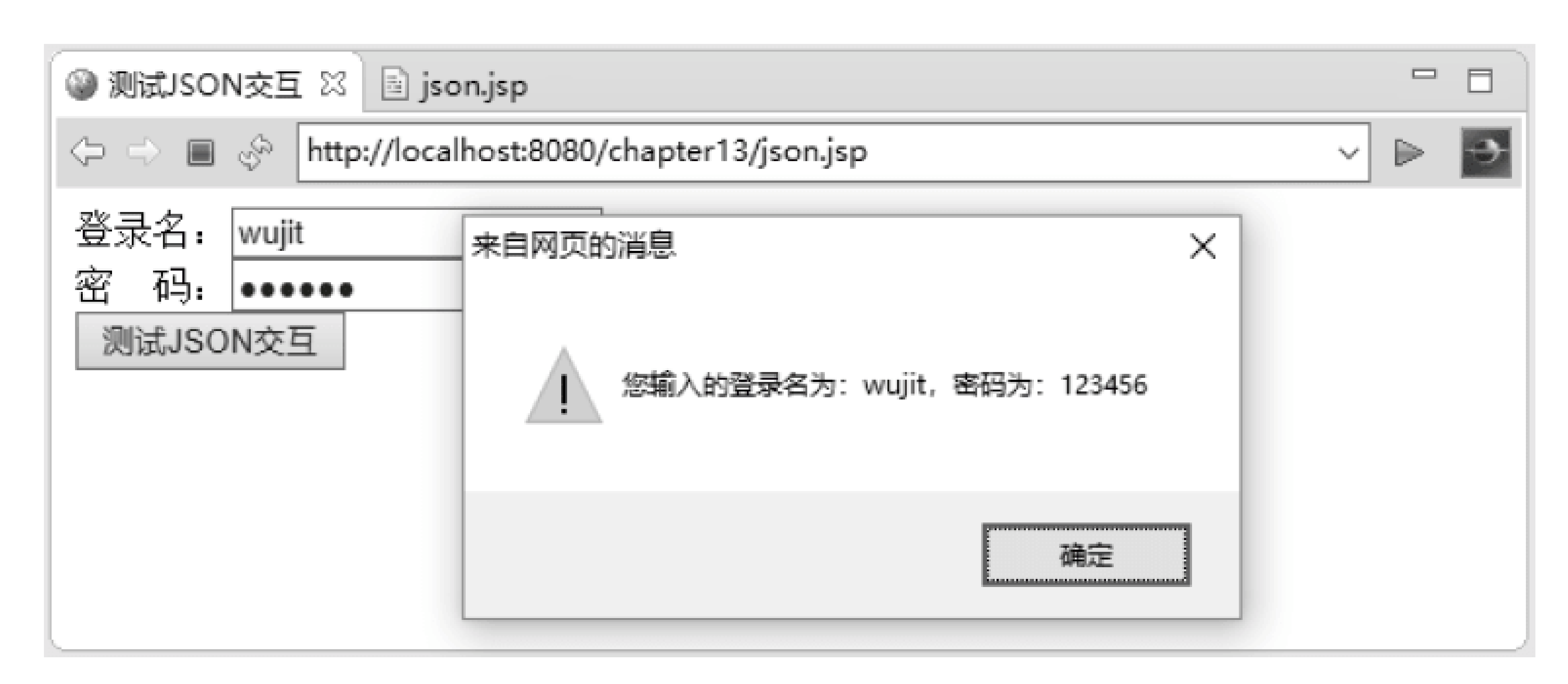


图 13.3 运行结果

从图 13.2 和图 13.3 的显示结果可以看出，编写的代码已经正确实现了 JSON 数据交互，可以将 JSON 格式的请求数据转换为方法中的 Java 对象，也可以将 Java 对象转换为 JSON 格式的响应数据。

13.2 RESTful 支持

Spring MVC 除了支持 JSON 数据交互外，还支持 RESTful 风格的编程。本节将介绍 RESTful 的概念和使用。

13.2.1 什么是 RESTful

RESTful 也称为 REST（Representational State Transfer），可以将它理解为一种软件架构风格或设计风格。

RESTful 风格就是把请求参数变成请求路径的一种风格。例如，传统的 URL 请求格式如下：

```
http://.../queryitems?id=1
```

而采用 RESTful 风格后，其 URL 请求为：

```
http://.../items/1
```

从上述两个请求中可以看出，RESTful 风格中的 URL 将请求参数 id=1 变成了请求路径的一部分，并且 URL 中的 queryItems 也变成了 items（RESTful 风格中的 URL 不存在动词形式的路径，如 queryItems 表示查询订单，是一个动词，而 tems 表示订单，为名词）。

RESTful 风格在 HTTP 请求中使用 put、delete、post 和 get 方式分别对应添加、删除、修改和查询的操作。不过目前国内开发还是只使用 post 和 get 方式进行增、删、改、查操作。

13.2.2 应用案例——查询客户信息

本案例将采用 RESTful 风格的请求实现对客户信息的查询，同时返回 JSON 格式的数据。具体

实现步骤如下。

步骤01 在控制器类 CustomerController 中编写客户查询方法 selectCustomer(),代码如下所示。

```
/*
* 接收 RESTful 风格的请求，其接收方式为 GET
*/
@RequestMapping(value="/customer/{id}",method=RequestMethod.GET)
@ResponseBody
public Customer selectCustomer(@PathVariable("id") Integer id){
    //查看接收数据
    System.out.println(id);
    Customer customer=new Customer();
    //模拟根据 id 查询出客户对象数据
    if(id==10){
        customer.setLoginname("wujit");
    }
    //返回 JSON 格式的数据
    return customer;
}
```

在上述代码中，@RequestMapping(value="customer/{id}", method= RequestMethod.GET)注解用于匹配请求路径（包括参数）和方式。其中 value="/user/{id}"表示可以匹配以“/user/{id}”结尾的请求，id 为请求中的动态参数；method= RequestMethod.GET 表示只接收 GET 方式的请求。方法中的@PathVariable("id")注解则用于接收并绑定请求参数，它可以将请求 URL 中的变量映射到方法的形参上，如果请求路径为“/user/{id}”，即请求参数中的 id 和方法形参名称 id 一样，@PathVariable 后面的“("id")”就可以省略。

步骤02 在 WebContent 目录下编写页面文件 restful.jsp,在页面中使用 AJAX 方式通过输入的客户编号来查询客户信息，如文件 13.6 所示。

文件 13.6 restful.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>RESTful 测试</title>
    <script type="text/javascript"
      src="${pageContext.request.contextPath }/js/jquery-1.11.3.min.js">
    </script>
    <script type="text/javascript">
    function search(){
```

```
        //获取输入的查询编号
        var id=$("#number").val();
        $.ajax({
            url:"${pageContext.request.contextPath }/customer/"+id,
            type:"GET",
            //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
            dataType:"json",
            //成功响应的结果
            success:function(data){
                if(data.loginname!=null){
                    alert("您查询的客户登录名为："+data.loginname);
                }else{
                    alert("没有找到 id 为："+id+"的客户！");
                }
            }
        });
    }
    </script>
    </head>
    <body>
     <form>
     客户编号：<input type="text" name="number" id="number" /> <br />
     <input type="button" value="查询" onclick="search()" />
     </form>
    </body>
</html>
```

在文件 13.6 中，在请求路径中使用了 RESTful 风格的 URL，并且定义了请求方式为 GET。

步骤03 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter13/restful.jsp，其显示效果如图 13.4 所示。

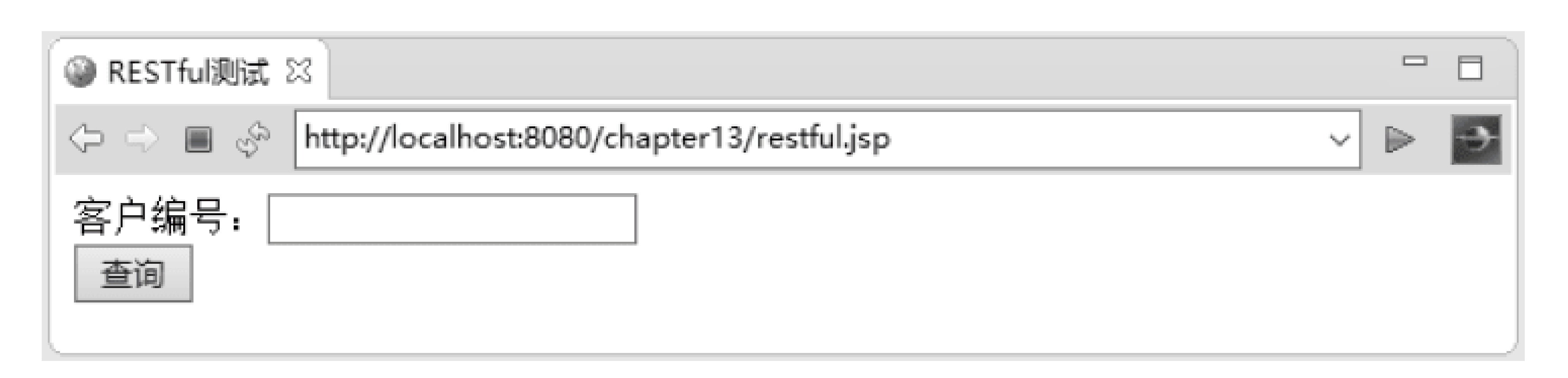


图 13.4　restful. jsp 测试页面

在输入框中输入编号“10”后，单击“查询”按钮，程序正确执行后，浏览器会弹出客户信息窗口，如图 13.5 所示。

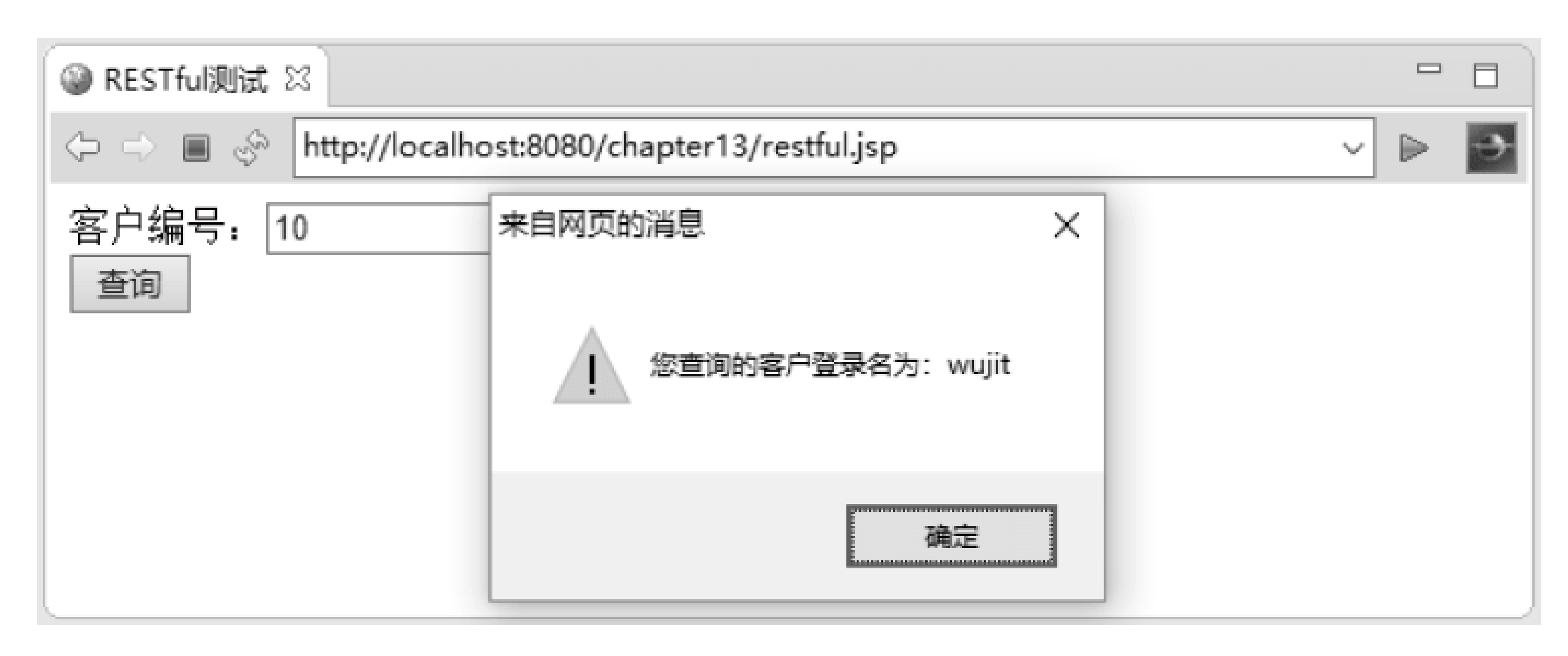


图 13.5　运行结果

从图 13.4 和图 13.5 的显示结果可以看出，已经成功地使用 RESTful 风格的请求查询出了客户信息。

13.3　习　题

1. 请简要概述什么是 JSON？JSON 有哪两种数据结构？

2. 参照本章案例对 Spring MVC 中的 JSON 数据交互和 RESTful 风格进行实践，并仔细分析其执行过程。

第 14 章

拦截器

拦截器的使用是非常普遍的。例如在 OA 系统中通过拦截器可以拦截未登录的用户，或者使用它来验证已登录用户是否有相应的操作权限等。Spring MVC 中提供了拦截器功能，通过配置即可对请求进行拦截处理。本章将讲解 Spring MVC 中拦截器的使用。

本章主要涉及的知识点如下。

- 拦截器定义和配置方式。
- 熟悉拦截器的执行流程。
- 拦截器的使用。

14.1 拦截器概述

Spring MVC 中的拦截器（Interceptor）类似于 Servlet 中的过滤器（Filter），它主要用于拦截用户请求并做相应的处理。例如通过拦截器可以进行权限验证、判断用户是否已登录等。

14.1.1 拦截器的定义

要使用 Spring MVC 中的拦截器，就需要对拦截器类进行定义和配置。通常拦截器类可以通过两种方式来定义：

- 一种是通过实现 HandlerInterceptor 接口或者继承 HandlerInterceptor 接口的实现类（如 HandlerInterceptorAdapter）来定义。
- 另一种是通过实现 WebRequestInterceptor 接口或继承 WebRequestInterceptor 接口的实现类来定义。

以实现 HandlerInterceptor 接口的定义方式为例，自定义拦截器类的代码如下所示。

```
public class UserInterceptor implements HandlerInterceptor{
    @Override
public boolean preHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2)
        throws Exception {
          return false;
    }
    @Override
public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
            ModelAndView arg3) throws Exception {
    }
    @Override
public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
            Exception arg3) throws Exception {
    }
}
```

从上述代码可以看出，自定义的拦截器类实现了 HandlerInterceptor 接口，并实现了接口中的 3 个方法。关于这 3 个方法的具体描述如下。

- preHandle()方法：该方法会在控制器方法前执行，其返回值表示是否中断后续操作。当其返回值为 true 时，表示继续向下执行；当其返回值为 false 时，会中断后续的所有操作（包括调用下一个拦截器和控制器类中的方法执行等）。
- postHandle()方法：该方法会在控制器方法调用之后，且解析视图之前执行。可以通过此方法对请求域中的模型和视图做出进一步的修改。
- afterCompletion()方法：该方法在整个请求完成，即视图渲染结束之后执行。可以通过此方法实现一些资源清理、记录日志信息等工作。

14.1.2　拦截器的配置

要使自定义的拦截器类生效，需要在 Spring MVC 的配置文件中进行配置，配置代码如下所示。

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <!-- 使用 bean 直接定义在<mvc:interceptors>下面的 Interceptor 将拦截所有请求 -->
    <bean class="com.ssm.interceptor.UserInterceptor" />
    <!-- 拦截器 1 -->
    <mvc:interceptor>
        <!-- 配置拦截器作用的路径 -->
        <mvc:mapping path="/**" />
        <!-- 配置不需要拦截器作用的路径 -->
        <mvc:exclude-mapping path="" />
        <!-- 定义在<mvc:interceptor>下面，表示对匹配路径的请求才进行拦截 -->
        <bean class="com.ssm.interceptor.Interceptor1" />
```

```
        </mvc:interceptor>
        <!-- 拦截器 2 -->
        <mvc:interceptor>
            <mvc:mapping path="/hello" />
            <bean class="com.ssm.interceptor.Interceptor2" />
        </mvc:interceptor>
            ......
    </mvc:interceptors>
```

在上述代码中，<mvc:interceptors>元素用于配置一组拦截器，其子元素<bean>中定义的是全局拦截器，它会拦截所有的请求；而<mvc:interceptor>元素中定义的是指定路径的拦截器，它会对指定路径下的请求生效。<mvc:interceptor>元素的子元素<mvc:mapping>用于配置拦截器作用的路径，该路径在其属性 path 中定义。如上述代码中 path 的属性值"/**"表示拦截所有路径，"/hello"表示拦截所有以"hello"结尾的路径。如果在请求路径中包含不需要拦截的内容，还可以通过<mvc:exclude-mapping>元素进行配置。

注　意

<mvc: interceptor>中的子元素必须按照上述代码的配置顺序进行编写，即<mvc: mapping…/>→<mvc: exclude-mapping…/>→<bean…/>的顺序，否则文件会报错。

14.2　拦截器的执行流程

拦截器的执行是有一定顺序的，该顺序与配置文件中所定义的拦截器的顺序相关。本节将对单个拦截器的执行流程和多个拦截器的执行流程进行讲解。

14.2.1　单个拦截器的执行流程

如果在项目中只定义了一个拦截器，那么该拦截器在程序中的执行流程如图 14.1 所示。从中可以看出，程序首先会执行拦截器类中的 preHandle()方法，如果该方法的返回值为 true，则程序就会继续向下执行处理器中的方法，否则将不再向下执行；在业务处理器（即控制器 Controller 类）处理完请求后，会执行 postHandle()方法，然后通过 DispatcherServlet 向客户端返回响应；在 DispatcherServlet 处理完请求后，才会执行 afterCompletion()方法。

【示例 14-1】下面通过一个案例来演示拦截器的执行流程。

步骤01 在 Eclipse 中创建一个名为 chapter14 的 Web 项目，将 Spring MVC 程序运行所需的 JAR 包复制到项目的 lib 目录中，并发布到类路径下。

步骤02 在 web.xml 中配置 Spring MVC 的前端过滤器和初始化加载配置文件等信息。

步骤03 在 src 目录下创建一个 com.ssm.controller 包，并在包中创建控制器类 HelloController，编辑后的代码如文件 14.1 所示。

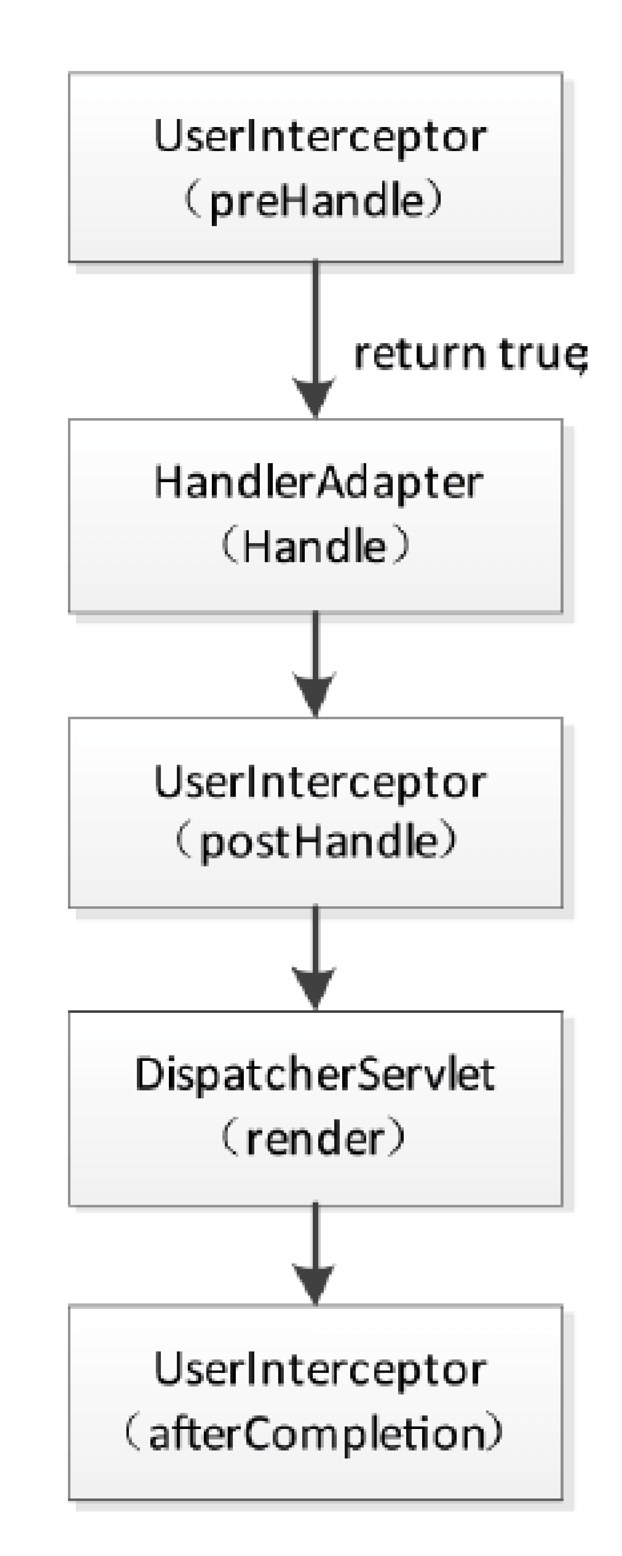


图 14.1　单个拦截器的执行流程

文件 14.1　HelloController.java

```
package com.ssm.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HelloController {
    @RequestMapping("/hello")
    public String hello(){
        System.out.println("Hello");
        return "success";
      }
}
```

步骤04 在 src 目录下创建一个 com.ssm.interceptor 包，并在包中创建拦截器类 UserInterceptor。该类需要实现 HandlerInterceptor 接口，并且在实现方法中需要编写输出语句来输出信息，如文件 14.2 所示。

文件 14.2　UserInterceptor.java

```
package com.ssm.interceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
public class UserInterceptor implements HandlerInterceptor{
    @Override
    public boolean preHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object
    arg2)
    throws Exception {
```

```
        System.out.println("UserInterceptor...preHandle");
        //对拦截的请求进行放行处理
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
        ModelAndView arg3) throws Exception {
        System.out.println("UserInterceptor...postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
        Exception arg3) throws Exception {
        System.out.println("UserInterceptor...afterCompletion");
    }
}
```

步骤05 在 src 目录下创建并配置 Spring MVC 的配置文件，如文件 14.3 所示。

文件 14.3 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!--指定需要扫描的包 -->
    <context:component-scan base-package="com.ssm.controller" />
    <!-- 定义视图解析器 -->
    <bean id="viewResoler"
     class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     <!-- 设置前缀 -->
     <property name="prefix" value="/WEB-INF/jsp/" />
     <!-- 设置后缀 -->
     <property name="suffix" value=".jsp" />
    </bean>
    <!-- 配置拦截器 -->
    <mvc:interceptors>
     <!-- 使用 bean 直接定义在<mvc:interceptors>下面的 Interceptor 将拦截所有请求 -->
     <bean class="com.ssm.interceptor.UserInterceptor" />
    </mvc:interceptors>
</beans>
```

由于配置拦截器使用的是<mvc: interceptors>元素，因此需要配置 mvc 的 schema 信息。本案例演示的是单个拦截器的执行顺序，所以这里只配置了一个全局的拦截器。

步骤06 在 WEB-INF 目录下创建一个 jsp 文件夹，并在该文件夹中创建一个页面文件 success.jsp，然后在页面文件的<body>元素内编写任意显示信息，如“ok，执行成功!”。

步骤07 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter14/hello，程序正确执行后，浏览器会跳转到 success.jsp 页面，此时控制台的输出结果如图 14.2 所示。从图 14.2 中可以看出，程序先执行了拦截器类中的 preHandle()方法，然后执行了控制器中的 Hello()方法，最后分别执行了拦截器类中的 postHandle()方法和 afterCompletion()方法，这与前文描述的单个拦截器的执行顺序是一致的。

```
Markers  Properties  Servers  Snippets  Problems  Console  JUnit  Terminal
Tomcat v7.0 Server at localhost [Apache Tomcat] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年11月12日 下午9:36:06)
UserInterceptor...preHandle
Hello
UserInterceptor...postHandle
UserInterceptor...afterCompletion
```

图 14.2　运行结果

14.2.2　多个拦截器的执行流程

在大型项目中，通常会定义很多拦截器来实现不同的功能。多个拦截器的执行顺序如图 14.3 所示。这里假设有两个拦截器 Interceptor1 和 Interceptor2，并且在配置文件中，Interceptor1 拦截器配置在前。

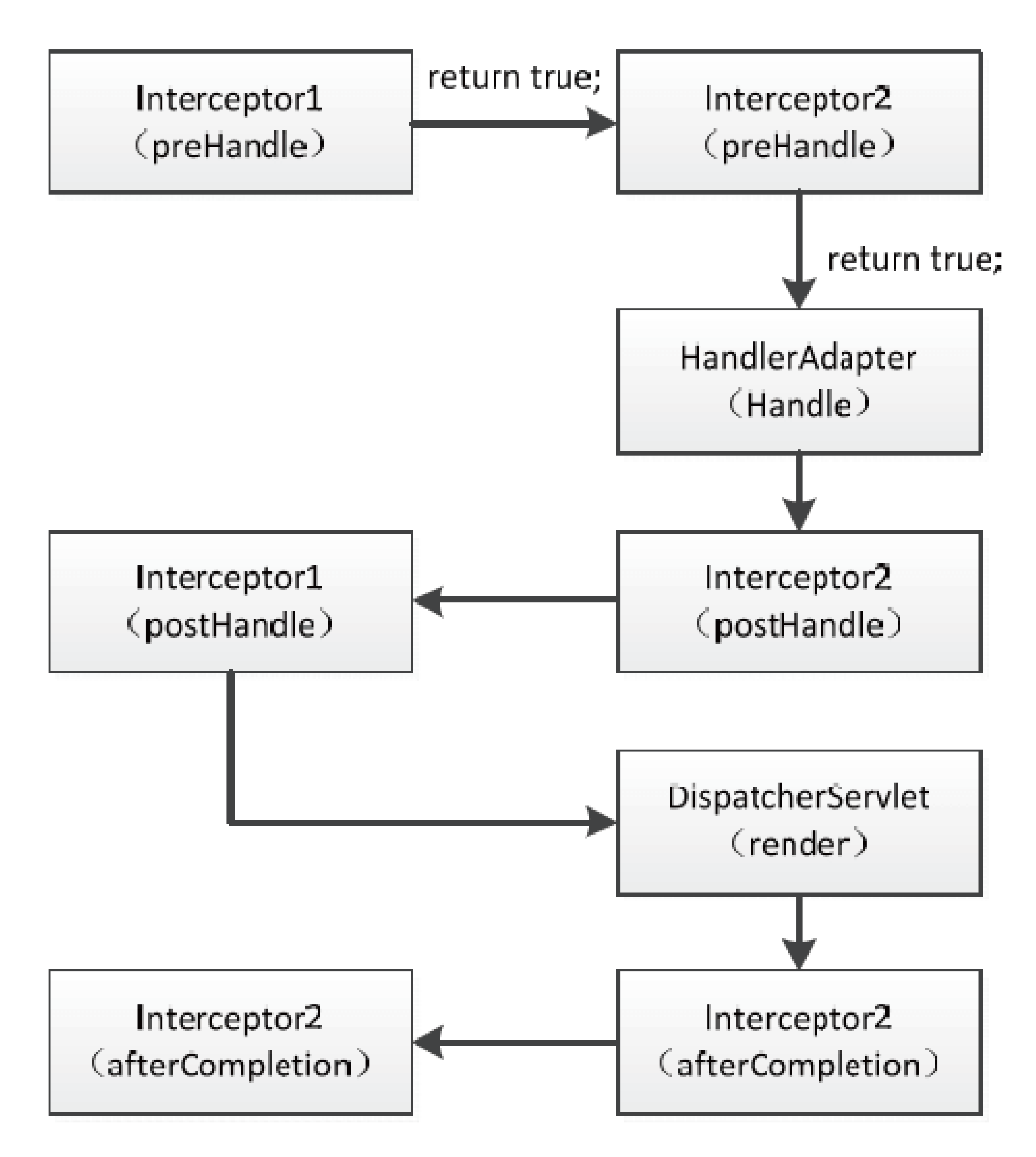


图 14.3　多个拦截器的执行流程

从图 14.3 可以看出，当有多个拦截器同时工作时，它们的 preHandle()方法会按照配置文件中拦截器的配置顺序执行，而它们的 postHandle()方法和 afterCompletion()方法则会按照配置顺序的反序执行。

【示例 14-2】为了验证上述描述，下面修改 14.2.1 小节的案例来演示多个拦截器的执行。

步骤01 在 com.ssm.interceptor 包中创建两个拦截器类 Interceptor1 和 Interceptor2，这两个拦截器类均实现了 HandlerInterceptor 接口，并重写其中的方法，如文件 14.4 和文件 14.5 所示。

文件 14.4　Interceptor1.java

```java
package com.ssm.interceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
public class Interceptor1 implements HandlerInterceptor{
    @Override
    public boolean preHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2)
   throws Exception {
     System.out.println("UserInterceptor1...preHandle");
     return true;
    }
    @Override
    public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
       ModelAndView arg3)  throws Exception {
     System.out.println("UserInterceptor1...postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
         Exception arg3) throws Exception {
     System.out.println("UserInterceptor1...afterCompletion");
    }
}
```

文件 14.5　Interceptor2.java

```java
package com.ssm.interceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
public class Interceptor2 implements HandlerInterceptor{
    @Override
    public boolean preHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2)
   throws Exception {
     System.out.println("UserInterceptor2...preHandle");
     return true;
    }
    @Override
```

```
    public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object
    arg2,
        ModelAndView arg3)  throws Exception {
     System.out.println("UserInterceptor2...postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object
    arg2,
          Exception arg3) throws Exception {
     System.out.println("UserInterceptor2...afterCompletion");
    }
}
```

步骤02 在配置文件 springmvc-config.xml 中的<mvc:interceptors>元素内配置上面所定义的两个拦截器，配置代码如下所示。

```
<!-- 拦截器 1 -->
<mvc:interceptor>
    <!-- 配置拦截器作用的路径 -->
    <mvc:mapping path="/**" />
    <!-- 定义在<mvc:interceptor>下面，表示对匹配路径的请求才进行拦截 -->
    <bean class="com.ssm.interceptor.Interceptor1" />
</mvc:interceptor>
<!-- 拦截器 2 -->
<mvc:interceptor>
    <!-- 配置拦截器作用的路径 -->
    <mvc:mapping path="/hello" />
    <!-- 定义在<mvc:interceptor>下面，表示对匹配路径的请求才进行拦截 -->
    <bean class="com.ssm.interceptor.Interceptor2" />
</mvc:interceptor>
```

在上述拦截器的配置代码中，第一个拦截器会作用于所有路径下的请求，而第二个拦截器会作用于以“/hello”结尾的请求。

注　意
为了不影响程序的输出结果，可先注释掉 14.2.1 小节案例中的配置。

步骤03 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter14/hello，程序正确执行后，浏览器会跳转到 success.jsp 页面，此时控制台的输出结果如图 14.4 所示。

从图 14.4 中可以看出，程序先执行了前两个拦截器类中的 preHandle()方法，这两个方法的执行顺序与配置文件中定义的顺序相同；然后执行了控制器类中的 helo()方法；最后执行了两个拦截器类中的 postHandle()方法和 afterCompletion()方法，且这两个方法的执行顺序与配置文件中所定义的拦截器顺序相反。

```
Markers  Properties  Servers  Snippets  Problems  Console  JUnit  Terminal
Tomcat v7.0 Server at localhost [Apache Tomcat] D:\Program Files (x86)\Java\jre7\bin\javaw.exe (2018年11月12日 下午10:20:48)
UserInterceptor1...preHandle
UserInterceptor2...preHandle
Hello
UserInterceptor2...postHandle
UserInterceptor1...postHandle
UserInterceptor2...afterCompletion
UserInterceptor1...afterCompletion
```

图 14.4 运行结果

14.3 应用案例——用户登录权限验证

本节将通过拦截器来完成一个用户登录权限验证的案例。该案例的整个执行流程如图 14.5 所示。

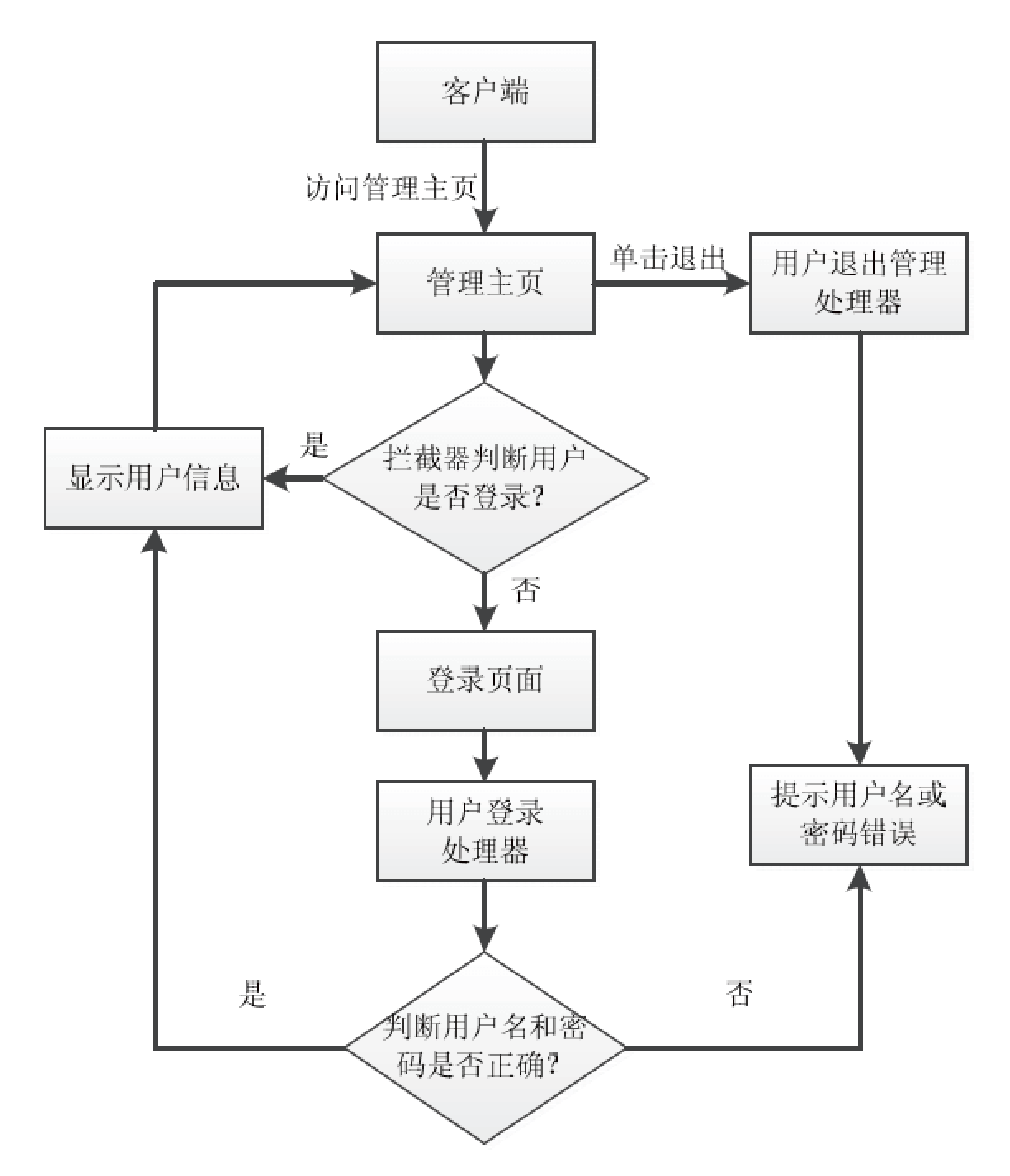


图 14.5 用户权限验证的执行流程图

从图 14.5 所示的流程图看出，只有登录后的用户才能访问管理主页，如果没有登录而直接访问主页，拦截器就会将请求拦截，并转发到登录页面，同时在登录页面中给出提示信息。如果用户名或密码错误，也会在登录页面给出相应的提示信息。当已登录的用户在管理主页中单击“退出”链接时，同样会回到登录页面。

接下来讲解如何在项目中实现用户登录权限验证，具体步骤如下。

步骤01 在 src 目录下创建一个 com.ssm.po 包，并在包中创建 User 类，如文件 14.6 所示。

文件 14.6　User.java

```
package com.ssm.po;
public class User {
    private Integer id;
    private String username;
    private String password;
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getUsername() {
     return username;
    }
    public void setUsername(String username) {
     this.username = username;
    }
    public String getPassword() {
     return password;
    }
    public void setPassword(String password) {
     this.password = password;
    }
}
```

步骤02 在 com.ssm.controller 包中创建控制器类 UserController，并在该类中定义向主页跳转、向登录页面跳转、执行用户登录等操作的方法，如文件 14.7 所示。

文件 14.7　UserController.java

```
package com.ssm.controller;
import javax.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.ssm.po.User;
@Controller
public class UserController {
    /*
     * 向用户登录页面跳转
     */
    @RequestMapping(value="/toLogin",method=RequestMethod.GET)
    public String toLogin(){
     return "login";
    }
```

```
        /*
         * 用户登录
         */
        @RequestMapping(value="/login",method=RequestMethod.POST)
        public String login(User user,Model model,HttpSession session){
         String username=user.getUsername();
         String password=user.getPassword();
         //模拟从数据库获取用户名和密码进行判断
         if(username!=null && username.equals("wujit")){
             if(password!=null && password.equals("123456")){
                //用户存在，将用户信息保存到 Session 中，并复位向到主页
                 session.setAttribute("user_session", user);
                 return "redirect:main";
             }
         }
         //用户不存在，添加错误信息到 model 中，并跳转到登录页面
         model.addAttribute("msg","用户名或密码错误，请重新输入！");
         return "login";
        }
        /*
         * 向管理主页跳转
         */
        @RequestMapping(value="/main")
        public String toMain(){
         return "main";
        }
        /*
         * 退出
         */
        @RequestMapping(value="/logout")
        public String logout(HttpSession session){
         session.invalidate();
         return "redirect:toLogin";
        }
    }
```

在文件 14.7 中，在用户登录方法中，先通过 User 类型的参数获取了用户名和密码，然后通过语句来模拟从数据库中获取到用户名和密码后进行判断。如果存在此用户，就将用户信息保存到 Session 中，并复位向到主页，否则跳转到登录页面。

步骤03 在 com.ssm.interceptor 包中创建拦截器类 LoginInterceptor，编辑后的代码如文件 14.8 所示。

文件 14.8 LoginInterceptor.java

```
package com.ssm.interceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import javax.servlet.http.HttpSession;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
import com.ssm.po.User;
public class LoginInterceptor implements HandlerInterceptor{
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
    Object handler) throws Exception {
    //获取请求的 URL
    String url=request.getRequestURI();
    //允许公开访问"/toLogin"
    if(url.indexOf("/toLogin")>=0){
        return true;
    }
    //允许公开访问"/login"
    if(url.indexOf("/login")>=0){
        return true;
    }
    //获取 session
    HttpSession session=request.getSession();
    User user=(User)session.getAttribute("user_session");
    //如果 user 不为空，表示已登录
    if(user!=null){
        return true;
    }
    //如果 user 为空，表示未登录
    request.setAttribute("msg","请先登录！");
    request.getRequestDispatcher("WEB-INF/jsp/login.jsp").forward(request, response);
    return false;
    }
    @Override
    public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object
    arg2,
    ModelAndView arg3) throws Exception {
    }

    @Override
    public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1,
    Object arg2, Exception arg3) throws Exception {
    }
}
```

在文件 14.8 的 preHandle()方法中，先获取了请求的 URL，然后通过 indexOf()方法判断 URL 中是否有“/toLogin”或“/login”字符串。如果有，就返回 true，即直接放行；如果没有，就继续向下执行拦截处理。接下来获取了 Session 中的用户信息，如果 Session 中包含用户信息，即表示用户已登录，就直接放行；否则会转发到登录页面，不再执行本方法中的后续程序。

步骤04 在配置文件的<mvc: interceptors>元素中配置自定义的登录拦截器信息，代码如下所示。

```
<mvc:interceptor>
    <mvc:mapping path="/**" />
    <bean class="com.ssm.interceptor.LoginInterceptor" />
</mvc:interceptor>
```

步骤05 在 WEB-INF 目录下的 jsp 文件夹中创建一个管理主页面 main.jsp。在该页面中，使用 EL 表达式获取用户信息，并通过一个超链接来实现“退出”功能，如文件 14.9 所示。

文件 14.9 main.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>管理主页</title>
 </head>
 <body>
        当前用户信息：${user_session.username}
         <a href="${pageContext.request.contextPath}/logout">退出</a>
 </body>
</html>
```

步骤06 在 WEB-INF 目录下的 jsp 文件夹中创建一个登录页面 login.jsp，在页面中编写一个用于实现登录操作的 form 表单，如文件 14.10 所示。

文件 14.10 login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 <title>用户登录</title>
 </head>
 <body>
     <form action="${pageContext.request.contextPath}/login" method="post">
    登录名：<input type="text" name="username" id="username" /> <br />
    密    码：
         <input type="password" name="password" id="password" /> <br />
    <input type="submit" value="登录" />
    </form>
```

```
    </body>
</html>
```

步骤07 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter14/main，程序正确执行后，其页面显示效果如图 14.6 所示。

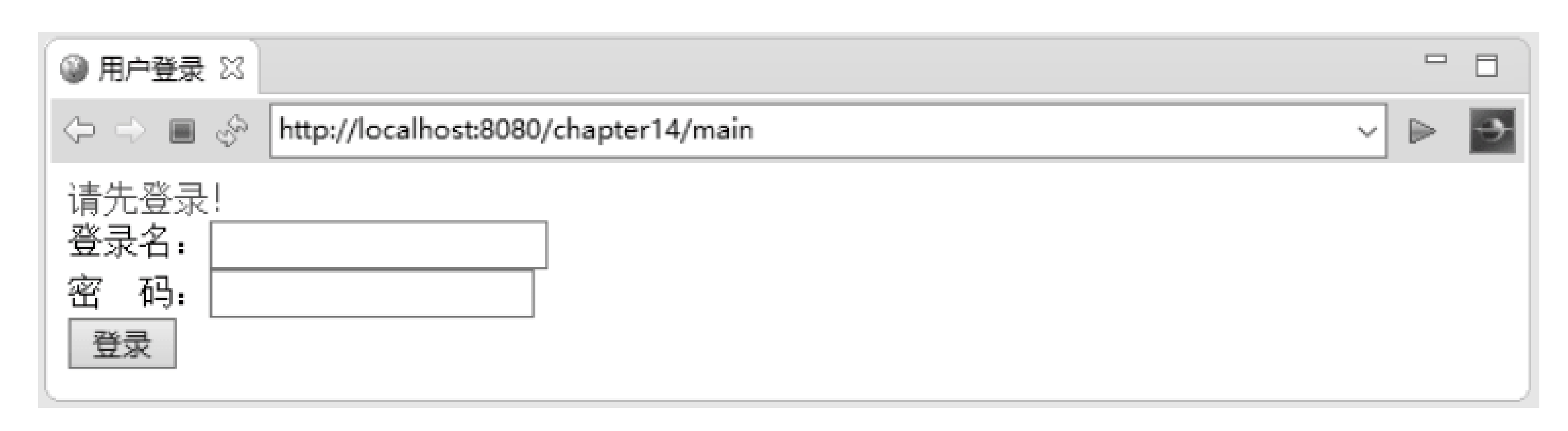


图 14.6　执行后 login.jsp 页面的显示结果

从图 14.6 可以看出，当用户未登录而直接访问主页面时，访问请求会被登录拦截器拦截，从而跳转到登录页面，并提示用户未登录信息。如果在用户名输入框中输入“zhangsan”，密码框中输入“123456”，当单击“登录”按钮后，浏览器的显示结果如图 14.7 所示。

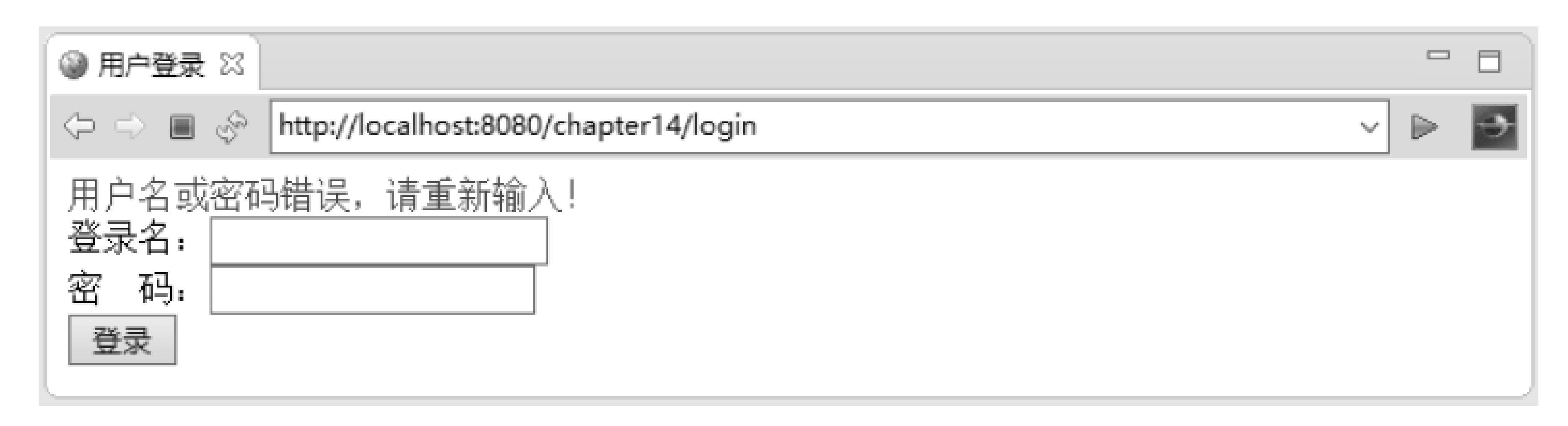


图 14.7　执行后 login.jsp 页面的显示结果

当输入正确的用户名“wujit”和密码“123456”，并单击“登录”按钮后，浏览器会跳转到管理主页面，如图 14.8 所示。当单击图 14.8 中的“退出”链接后，用户即可退出当前管理页面，复位向到登录页面。

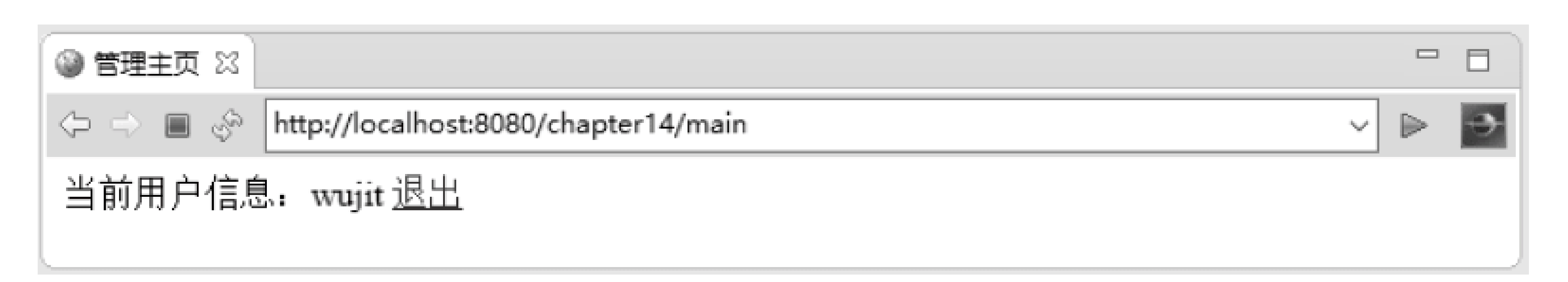


图 14.8　执行后 main.jsp 页面的显示结果

14.4　习　　题

1. 请简述单个拦截器和多个拦截器的执行流程，并通过案例进行演示。
2. 列举项目中拦截器的应用场合，并在实际项目中进行实践应用。

第 15 章

SSM 框架整合

前面已对 Spring、MyBatis、Spring MVC 三大框架以及 Spring 与 MyBatis 整合的使用进行了讲解。在实际项目开发中经常将三大框架一起整合使用。本章就对 SSM（Spring、Spring MVC、MyBatis）框架的整合使用进行讲解和应用测试。

本章主要涉及的知识点如下。

- SSM 框架的整合思路。
- SSM 框架整合的配置文件内容编写。
- SSM 框架整合应用程序的编写。

15.1 整合环境搭建

Spring MVC 是 Spring 框架中的一个模块，那么 Spring MVC 与 Spring 之间不需要整合，只需要引入相应 JAR 包就可以直接使用。SSM 框架的整合只需要在 Spring 与 MyBatis 之间和 Spring MVC 与 MyBatis 之间进行整合。

15.1.1 整合思路

SSM 框架的整合思路：

通过 Spring 实例化 Bean，然后调用实例对象中的查询方法来执行 MyBatis 映射文件中的 SQL 语句，如果能够正确查询出数据库中的数据，就可以认为 Spring 与 MyBatis 框架整合成功。

如果可以通过前台页面来执行查询方法，并且查询出的数据能够在页面中正确显示，那么可以认为 SSM 三大框架整合成功。

15.1.2　准备所需 JAR 包

要实现 SSM 框架的整合，需要准备这三个框架的 JAR 包，以及其他整合所需的 JAR 包。在前面讲解 Spring 与 MyBatis 框架整合时，已经介绍了 Spring 与 MyBatis 整合所需要的 JAR 包，这里只需要再加入 Spring MVC 的相关 JAR 包即可，具体如下。

- spring-web-4.3.6.RELEASE.jar
- spring-webmvc-4.3.6.RELEASE.jar

SSM 整合时所需的全部 JAR 包如图 15.1 所示。



图 15.1　SSM 整合 JAR 包

15.1.3　编写配置文件

步骤01 在 Eclipse 中创建一个名为 chapter15 的 Web 项目，将整合所需的 JAR 包添加到项目的 lib 目录中，并发布到类路径下。

步骤02 在项目 src 目录下分别创建数据库常量配置文件 db.properties、Spring 配置文件 applicationContext.xml 以及 MyBatis 的配置文件 mybatis-config.xml。这 3 个配置文件的实现代码如文件 15.1、文件 15.2 和文件 15.3 所示。

文件 15.1 db.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/db_mybatis
jdbc.username=root
jdbc.password=root
jdbc.maxTotal=30
jdbc.maxIdle=10
jdbc.initialSize=5
```

文件 15.2 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!--读取 db.properties-->
    <context:property-placeholder location="classpath:db.properties"/>
    <!--配置数据源 -->
    <bean id="dataSource"
     class="org.apache.commons.dbcp2.BasicDataSource">
     <!--数据库驱动 -->
     <property name="driverClassName" value="${jdbc.driver}" />
     <!--连接数据库的 url -->
     <property name="url" value="${jdbc.url}" />
     <!--连接数据库的用户名 -->
     <property name="username" value="${jdbc.username}" />
     <!--连接数据库的密码-->
     <property name="password" value="${jdbc.password}" />
     <!--最大连接数-->
     <property name="maxTotal" value="${jdbc.maxTotal}" />
     <!--最大空闲连接-->
     <property name="maxIdle" value="${jdbc.maxIdle}" />
     <!--初始化连接数-->
     <property name="initialSize" value="${jdbc.initialSize}" />
    </bean>
    <!--事务管理器，依赖于数据源 -->
    <bean id="transactionManager"
```

```
            class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!--注册事务管理器驱动，开启事务注解 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>
    <!--配置 MyBatis 工厂 -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--注入数据源 -->
        <property name="dataSource" ref="dataSource" />
        <!--指定核心配置文件位置 -->
        <property name="configLocation" value="classpath:mybatis-config.xml" />
    </bean>
    <!--配置 mapper 扫描器 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" ref="com.ssm.dao"></property>
    </bean>
    <!--扫描 Service -->
    <context:component-scan base-package="com.ssm.service"/>
</beans>
```

在文件 15.2 中，首先定义了读取 db.properties 文件的配置和数据源配置，然后配置了事务管理器并开启了事务注解。接下来配置了用于整合 MyBatis 框架的 MyBatis 工厂信息，最后定义了 mapper 扫描器来扫描 DAO 层以及扫描 Service 层的配置。

> **注　意**
>
> 在实际开发时，为了避免 Spring 配置文件中的信息过于臃肿，通常会将 Spring 配置文件中的信息按照不同的功能分散在多个配置文件中。例如可以将事务配置放置在名称为 applicationContext-transaction.xml 的文件中，将数据源等信息放置在名称为 applicationContext-db.xml 的文件中等。在 web.xml 中配置加载 Spring 文件信息时，只需通过 applicationContext-*.xml 的方式即可自动加载全部配置文件。

文件 15.3　mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--配置别名 -->
    <typeAliases>
        <package name="com.ssm.po"/>
    </typeAliases>
</configuration>
```

由于在 Spring 配置文件中已经配置了数据源信息以及 mapper 接口文件扫描器，因此在 MyBatis 的配置文件中只需要根据 POJO 类路径进行别名配置即可。

步骤03 在 config 文件夹中创建 Spring MVC 的配置文件 springmvc-config.xml，编辑后如文件 15.4 所示。

文件 15.4 springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!--指定需要扫描的包 -->
    <context:component-scan base-package="com.ssm.controller" />
    <!-- 配置注解驱动 -->
    <mvc:annotation-driven />
    <!-- 定义视图解析器 -->
    <bean id="viewResoler"
     class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     <!-- 设置前缀 -->
     <property name="prefix" value="/WEB-INF/jsp/" />
     <!-- 设置后缀 -->
     <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

在文件 15.4 中，主要配置了用于扫描@Controller 注解的包扫描器、注解驱动器以及视图解析器。

步骤04 在 web.xml 中配置 Spring 的文件监听器、编码过滤器以及 Spring MVC 的前端控制器等信息，如文件 15.5 所示。

文件 15.5 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
      id="WebApp_ID" version="3.0">
    <!-- 配置加载 Spring 文件的监听器 -->
    <context-param>
     <param-name>contextConfigLocation</param-name>
```

```
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- 编码过滤器 -->
    <filter>
        <filter-name>encoding</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>encoding</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
    <!-- 配置 Spring MVC 前端控制器 -->
    <servlet>
        <!-- 配置前端过滤器 -->
        <servlet-name>springmvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 初始化时加载配置文件 -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <!-- 配置服务器启动时立即加载 Spring MVC 配置文件 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <!-- /：拦截所有请求，jsp 除外 -->
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

15.2 整 合 测 试

15.1 节完成了 SSM 三大框架的整合工作。接下来，以查询学生信息为例讲解 SSM 框架的整合开发，其具体实现步骤如下。

步骤01 在 src 目录下创建一个 com.ssm.po 包，并在包中创建持久化类 User，如文件 15.6 所示。

文件 15.6 User.java

```
package com.ssm.po;
public class User {
    private Integer id;
    private String username;
    //其他属性省略
    public Integer getId() {
     return id;
    }
    public void setId(Integer id) {
     this.id = id;
    }
    public String getUsername() {
     return username;
    }
    public void setUsername(String username) {
     this.username = username;
    }
}
```

在文件 15.6 中编写了一个用于映射数据库表 t_user 的用户持久化类，在类中分别定义了 id、username、password 属性，以及其对应的 getter()/setter()方法。

步骤02 在 src 目录下创建一个 com.ssm.dao 包，并在包中创建接口文件 UserDao 以及对应的映射文件 UserDao.xml，如文件 15.7 和文件 15.8 所示。

文件 15.7 UserDao.java

```
package com.ssm.dao;
import com.ssm.po.User;
/*
 * User 接口文件
 */
public interface UserDao {
    /*
     * 根据 id 查询用户信息
     */
    public User findUserById(Integer id);
}
```

从上述代码可以看出，UserDao 中只定义了一个根据 id 查询客户信息的方法。

文件 15.8 UserDao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.dao.UserDao">
    <!--根据用户编号获取用户信息 -->
    <select id="findUserById" parameterType="Integer" resultType="User">
     select * from t_user where id=#{id}
    </select>
</mapper>
```

在文件 15.8 中，根据文件 15.7 中接口文件的方法编写了对应的执行语句信息。

> **注 意**
>
> 在前面整合环境搭建时，已经在配置文件 applicationContext.xml 中使用包扫描的形式加入了扫描包 com.ssm.dao 下的所有接口及映射文件，所以在这里完成 DAO 层接口及映射文件开发后，就不必再进行映射文件的扫描配置了。

步骤03 在 src 目录下创建一个 com.ssm.service 包，然后在包中创建接口文件 UserService，并在 UserService 中定义通过 id 查询客户的方法，如文件 15.9 所示。

文件 15.9 UserService.java

```
package com.ssm.service;
import com.ssm.po.User;
public interface UserService {
    public User findUserById(Integer id);
}
```

步骤04 在 src 目录下创建一个 com.ssm.service.impl 包，并在包中创建 UserService 接口的实现类 UserServiceImpl，如文件 15.10 所示。

文件 15.10 UserServiceImpl.java

```
package com.ssm.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.ssm.dao.UserDao;
import com.ssm.po.User;
import com.ssm.service.UserService;
//使用@Service 注解标识业务层的实现类
@Service
//使用@Transactional 注解标识类中的所有方法纳入 Spring 的事务管理
@Transactional
public class UserServiceImpl implements UserService {
```

```
    //注解注入 UserDao
    @Autowired
    private UserDao userDao;
    //查询客户
    public User findUserById(Integer id) {
     return this.userDao.findUserById(id);
    }
}
```

在文件 15.10 中，使用@Service 注解来标识业务层的实现类，使用@Transactional 注解来标识类中的所有方法都纳入 Spring 的事务管理，并使用@Autowired 注解将 UserDao 接口对象注入本类中，然后在本类的查询方法中调用 UserDao 对象的查询用户方法。

注　意

在上述代码中，@Transactional 注解主要是针对数据的增加、修改、删除进行事务管理的，上面示例中的查询方法并不需要使用该注解，此处的作用就是告知读者该注解在实际开发中应该如何使用。

步骤05 在 src 目录下创建一个 com.ssm.controller 包，并在包中创建用于处理页面请求的控制器类 UserController，如文件 15.11 所示。

文件 15.11　UserController.java

```
package com.ssm.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import com.ssm.po.User;
import com.ssm.service.UserService;
//@Controller 注解控制类
@Controller
public class UserController {
//@Autowired 注解注入
    @Autowired
    private UserService userService;
    /*
     * 根据 id 查询用户详情
     */
    @RequestMapping("/findUserById")
    public String findUserById(Integer id,Model model){
     User user=userService.findUserById(id);
     model.addAttribute("user", user);
     //返回用户信息展示页面
     return "user";
    }
```

```
}
```

在文件 15.11 中，先使用 Spring 的注解@Controller 来标识控制器类，然后通过@Autowired 注解将 UserService 接口对象注入本类中，最后编写了一个根据 id 查询用户详情的方法 findUserById()，该方法会将获取的用户详情返回到视图名为 user 的 JSP 页面中。

步骤06 在 WEB-INF 目录下创建一个 jsp 文件夹，在该文件夹下创建一个用于展示客户详情的页面文件 user.jsp，如文件 15.12 所示。

文件 15.12　user.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>用户信息</title>
</head>
<body>
    <table>
     <tr>
         <td>用户 ID</td>
         <td>${user.id}</td>
     </tr>
     <tr>
         <td>用户姓名</td>
         <td>${user.username}</td>
     </tr>
    </table>
</body>
</html>
```

在文件 15.12 中编写了一个用于展示用户信息的表格，表格会通过 EL 表达式来获取后台控制层返回的用户信息。

步骤07 发布并启动项目，在浏览器中访问地址 http://localhost:8080/chapter15/findUserById?id=1，其显示效果如图 15.2 所示。从中可以看出，通过浏览器已经成功查询出了 t_user 表中 id 为 1 的客户信息，这说明 SSM 框架整合成功了。

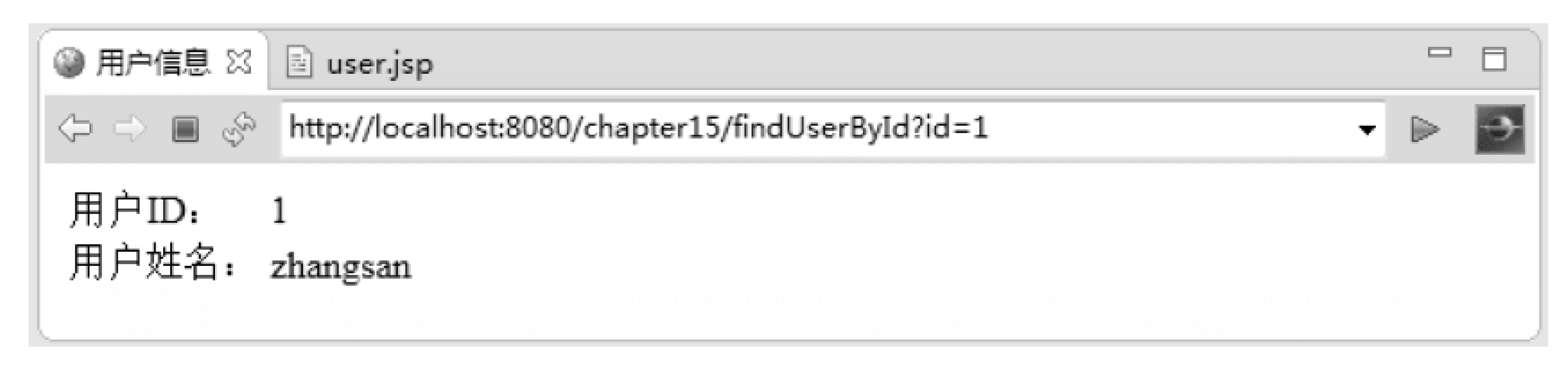


图 15.2　查询结果

注　意

在实际项目开发中，不单纯是查询信息，还存在包括增加、修改和删除在内的各种复杂业务。本节的案例只是测试 SSM 3 个框架的整合，即整合和测试 3 个框架是否能够“协同工作”。在第 16 章中，将通过一个综合性的案例——新闻发布管理系统来真正体验 SSM 三大框架的魅力。

15.3 习　　题

1. 请简述 SSM 框架的整合思路。
2. 对本章中的整合测试案例进行上机实践和测试。

第 16 章

SSM 实战：新闻发布管理系统

本章将通过前面学习的 SSM（Spring+Spring MVC+MyBatis）框架知识来实现一个新闻发布管理系统。该系统在开发过程中整合三大框架的基础上实现了系统后台的用户管理、用户登录、登录验证、新闻发布管理等。

本章主要涉及的知识点如下。

- 系统架构和文件组织结构。
- 数据分析与设计。
- 系统环境搭建。
- 系统功能设计和功能编码实现。

16.1　系统概述

本系统后台使用 SSM 三大框架实现，前台页面使用当前主流的 Bootstrap 和 jQuery 框架完成新闻展示。但系统前台实现中使用的 Bootstrap 和 jQuery 框架相关知识本书不做详细讲解。

16.1.1　系统功能需求

系统中主要实现了几大功能模块：用户管理（含角色设置和登录验证）、新闻类别管理、新闻发布管理和前台新闻展示等模块，如图 16.1 所示。

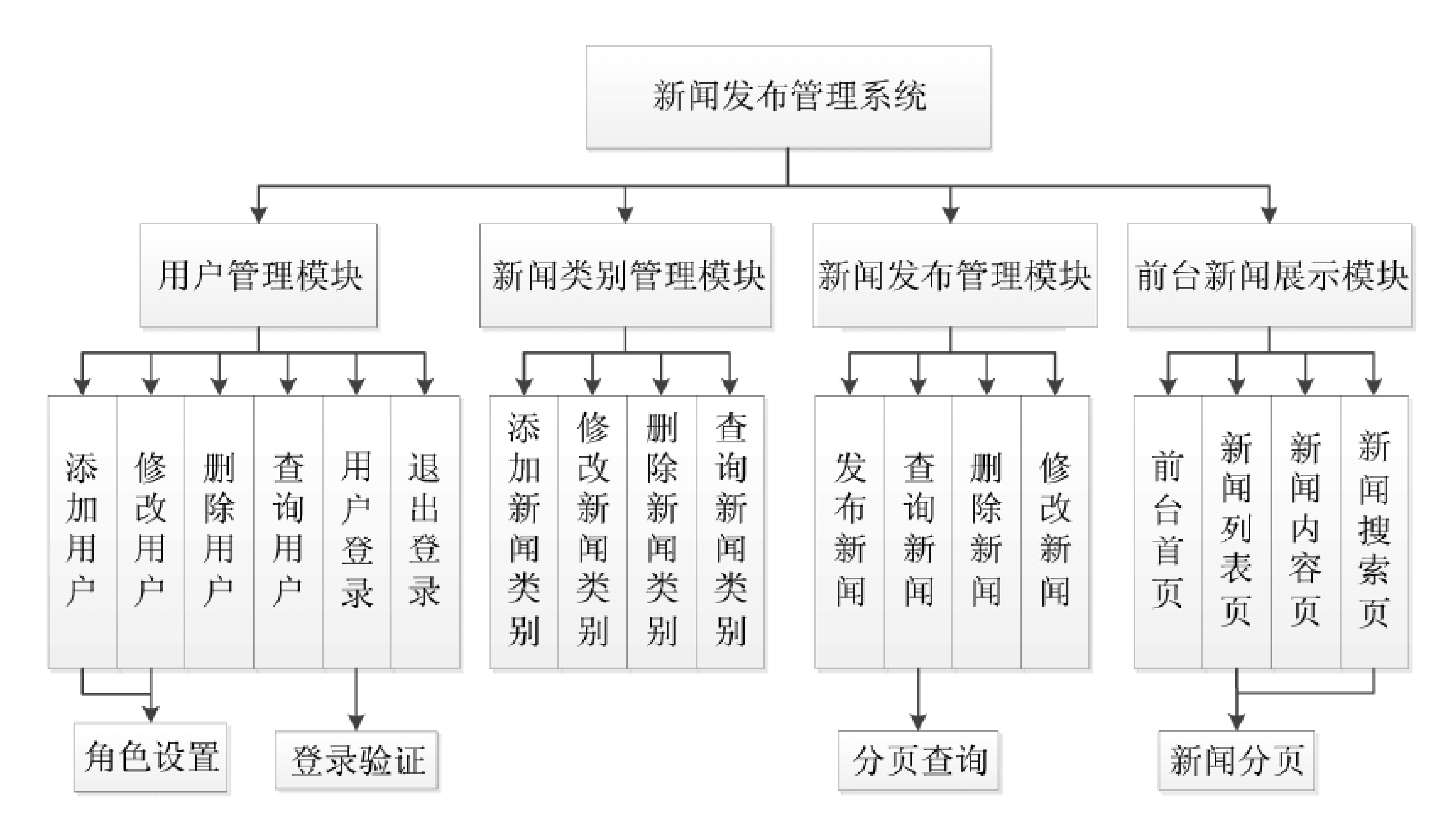


图 16.1 系统功能结构

16.1.2 系统架构设计

本系统根据功能的不同，项目结构可以划分为以下几个层次。

- 持久对象层：由若干持久化类（实体类）组成。
- 数据访问层：由若干 DAO 接口和 MyBatis 映射文件组成。接口的名称统一以 DAO 结尾，且 MyBatis 的映射文件名称要与接口的名称相同。
- 业务逻辑层：该层由若干 Service 接口和实现类组成。在本系统中，业务逻辑层的接口统一使用 Service 结尾，其实现类名称统一在接口名后加 Impl。该层主要用于实现系统的业务逻辑。
- Web 表现层：该层主要包括 Spring MVC 中的 Controller 类和 JSP 页面。Controller 类主要负责拦截用户请求，并调用业务逻辑层中相应组件的业务逻辑方法来处理用户请求，然后将相应的结果返回给 JSP 页面。

16.2 数据分析与设计

根据系统的功能需求，本系统的设计与实现中主要涉及角色实体、用户实体、新闻类别实体和新闻实体 4 个实体，其中角色实体与用户实体之间构成一对多的关联关系，新闻类别实体和新闻实体之间构成一对多的关联关系，用户实体和新闻实体之间构成一对多的关联关系。

与对象实体相适应，本系统中涉及角色表、用户表、新闻类别表和新闻表，其中用户表通过 roleId（角色 ID）字段与角色表构成关联关系，新闻表通过 CategoryId（类别 ID）字段与新闻类别表构成关联关系，新闻表还通过 userId（用户 ID）字段与用户表构成关联关系。这 4 张表的表结构如表 16.1、表 16.2、表 16.3 和表 16.4 所示。

表 16.1　角色表（t_role）

字段名	类型	长度	字段说明	备注
roleId	Int	32	角色 ID	主键
roleName	VarChar	20	角色名称	

表 16.2　用户表（t_user）

字段名	类型	长度	字段说明	备注
userId	Int	32	用户 ID	主键
userName	VarChar	20	用户姓名	
loginName	VarChar	20	登录账号	
password	VarChar	20	登录密码	
roleId	Int	32	角色 ID	外键
tel	VarChar	50	联系电话	
registerTime	DataTime		注册时间	
status	Char	1	注册状态	'1'：未启用 '2'：已启用 '3'：被禁用

表 16.3　新闻类别表（t_category）

字段名	类型	长度	字段说明	备注
categoryId	Int	32	类别 ID	主键
categoryName	VarChar	20	类别名称	

表 16.4　新闻表（t_news）

字段名	类型	长度	字段说明	备注
newsId	Int	32	类别 ID	主键
title	VarChar	60	信息标题	
contentTitle	VarChar	120	信息内容标题	
titlePicUrl	VarChar	120	标题图（路径）	
content	Text		信息内容	
contentAbstract	VarChar	300	内容摘要	
keywords	VarChar	100	关键词	
categoryId	Int	32	信息类别 ID	外键
userId	Int	32	发布用户 ID	外键
author	VarChar	30	作者（来源）	
publishTime	DataTime		发布时间	
clicks	Int	32	浏览次数	
publishStatus	Char	1	发布状态	'1'：发布 '2'：撤稿

本系统中使用 MySQL 数据库。在系统实现前，系统中的数据库资源需要提前准备好，创建数据库 news，并在数据库中创建上述 4 张表，同时添加一些必要的基础数据。创建数据库 news 的 SQL

语句和上述 4 个表的建表及基础数据插入的 SQL 语句如下所示。

```
# 创建数据库 news
CREATE DATABASE news;
# 使用数据库 news
USE news;
# 创建一个名称为 t_role 的表
CREATE TABLE t_role(
  roleId INT PRIMARY KEY,
  roleName VARCHAR(20)
);
# 插入两条数据
INSERT INTO t_role VALUES(1,'管理员');
INSERT INTO t_role VALUES(2,'信息员');
# 创建一个名称为 t_user 的表
CREATE TABLE t_user(
  userId INT PRIMARY KEY AUTO_INCREMENT,
  userName VARCHAR(20),
  loginName VARCHAR(20),
  password VARCHAR(20),
  tel VARCHAR(50),
  registerTime DATETIME,
  status CHAR(1),
  roleId INT ,
  FOREIGN KEY (roleId) REFERENCES t_role(roleId)
);
# 插入 1 条数据
INSERT INTO t_user(userName, loginName , password ,status,roleId)
            VALUES('无为','admin','123456','2',1);
# 创建一个名称为 t_category 的表
CREATE TABLE t_category(
  categoryId INT PRIMARY KEY,
  categoryName VARCHAR(20)
);
# 插入 4 条数据
INSERT INTO t_category VALUES(1,'今日头条');
INSERT INTO t_category VALUES(2,'综合资讯');
INSERT INTO t_category VALUES(3,'国内新闻');
INSERT INTO t_category VALUES(4,'国际新闻');
# 创建一个名称为 t_news 的表
CREATE TABLE t_news(
  newsId INT PRIMARY KEY AUTO_INCREMENT,
  title VARCHAR(60),
  contentTitle VARCHAR(120),
  titlePicUrl VARCHAR(120),
  content TEXT,
  contentAbstract VARCHAR(300),
```

```
    keywords VARCHAR(100),
    author VARCHAR(30),
    publishTime DATETIME,
    clicks INT,
    publishStatus CHAR(1),
    categoryId INT ,
    userId INT ,
    FOREIGN KEY (categoryId) REFERENCES t_category(categoryId),
    FOREIGN KEY (userId) REFERENCES t_user(userId)
);
```

16.3　系统功能设计与实现

本系统的设计与实现分模块进行，主要涉及：

- 开发环境和框架搭建
- 角色管理模块
- 用户管理模块
- 新闻类别管理模块
- 新闻发布管理模块
- 前台新闻展示模块

其中角色管理模块和新闻类别管理模块只通过创建对应的表格和插入初始化数据，并提供查询方法供用户管理和新闻发布管理等模块调用。前台新闻展示模块不做详细讲解，有兴趣的读者可以查看本章案例代码。

另外，从安全角度考虑，系统后台中使用拦截器对操作用户进行登录验证。

16.4　开发环境和框架搭建

系统的开发环境和框架搭建涉及各类文件的创建、引入和编写：包文件（内含各类接口和类）、配置文件、页面文件以及相关的 JAR 包文件、资源文件等。下面对系统开发环境和框架的搭建进行讲解。

16.4.1　创建项目，引入 JAR 包

在 Eclipse 中创建一个名称为 news_publish 的 Web 项目，将系统所准备的全部 JAR 包复制到项目的 lib 目录中，并发布到类路径下。

由于本系统使用 SSM 框架开发，因此需要准备三大框架的 JAR 包。另外，系统中还涉及数据

库连接、JSTL 标签等，所以还要准备其他包。整个系统中需要准备的 JAR 包共计 35 个，除了第 15 章 SSM 整合中用到的 29 个 JAR 包外，还包括以下 6 个 JAR 包。

- JSTL 标签库 JAR 包（两个）：taglibs-standard-spec-1.2.5.jar、taglibs-standard-impl-1.2.5.jar。
- Jackson 框架所需 JAR 包（3 个）：jackson-annotations-2.8.8.jar、jackson-core-2.8.8.jar、jackson-databind-2.8.8.jar。
- Java 工具类 JAR 包（1 个）：commons-lang3-3.4.jar。

16.4.2 编写配置文件

在项目 src 目录下分别创建数据库常量配置文件、Spring 配置文件、MyBatis 配置文件、Spring MVC 配置文件、log4j 配置文件以及资源配置文件。前 4 个文件如文件 16.1、文件 16.2、文件 16.3 和文件 16.4 所示。log4j 配置文件请参照前面相关章节的内容进行编写。资源配置文件本系统中没有用到。

文件 16.1 db.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/news
jdbc.username=root
jdbc.password=root
jdbc.maxTotal=30
jdbc.maxIdle=10
jdbc.initialSize=5
```

文件 16.2 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!--读取 db.properties-->
    <context:property-placeholder location="classpath:db.properties"/>
```

```
    <!--配置数据源 -->
    <bean id="dataSource"
     class="org.apache.commons.dbcp2.BasicDataSource">
     <!--数据库驱动 -->
     <property name="driverClassName" value="${jdbc.driver}" />
     <!--连接数据库的 url -->
     <property name="url" value="${jdbc.url}" />
     <!--连接数据库的用户名 -->
     <property name="username" value="${jdbc.username}" />
     <!--连接数据库的密码-->
     <property name="password" value="${jdbc.password}" />
     <!--最大连接数-->
     <property name="maxTotal" value="${jdbc.maxTotal}" />
     <!--最大空闲连接-->
     <property name="maxIdle" value="${jdbc.maxIdle}" />
     <!--初始化连接数-->
     <property name="initialSize" value="${jdbc.initialSize}" />
    </bean>
    <!--事务管理器，依赖于数据源 -->
    <bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
     <property name="dataSource" ref="dataSource"/>
    </bean>
    <!-- 通知 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
     <tx:attributes>
         <!-- 传播行为 -->
         <tx:method name="save*" propagation="REQUIRED"/>
         <tx:method name="insert*" propagation="REQUIRED"/>
         <tx:method name="add*" propagation="REQUIRED"/>
         <tx:method name="create*" propagation="REQUIRED"/>
         <tx:method name="delete*" propagation="REQUIRED"/>
         <tx:method name="update*" propagation="REQUIRED"/>
         <tx:method name="find*" propagation="SUPPORTS" read-only="true"/>
         <tx:method name="select*" propagation="SUPPORTS" read-only="true"/>
         <tx:method name="get*" propagation="SUPPORTS" read-only="true"/>
     </tx:attributes>
    </tx:advice>
    <!-- 切面 -->
    <aop:config>
     <aop:advisor advice-ref="txAdvice" pointcut="execution(* com.ssm.service.*.*(..))" />
    </aop:config>
    <!--配置 MyBatis 工厂 -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
     <!--注入数据源 -->
     <property name="dataSource" ref="dataSource" />
```

```
        <!--指定核心配置文件位置 -->
        <property name="configLocation" value="classpath:mybatis-config.xml" />
       </bean>
       <!--配置 mapper 扫描器 -->
       <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.ssm.dao"></property>
       </bean>
       <!--扫描 Service -->
       <context:component-scan base-package="com.ssm.service"/>
</beans>
```

applicationContext.xml 文件与第 15 章讲解 SSM 整合时有所不同的是，这里增加了事务传播行为以及切面的配置。在事务的传播行为中，只有查询方法的事务为只读，添加、修改和删除的操作都纳入了事务管理。

文件 16.3　mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--配置别名 -->
    <typeAliases>
     <package name="com.ssm.po"/>
    </typeAliases>
</configuration>
```

文件 15.4　springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!--指定需要扫描的包 -->
    <context:component-scan base-package="com.ssm.web.controller" />
    <!-- 配置注解驱动 -->
    <mvc:annotation-driven />
    <!-- 配置静态资源的访问映射，此配置中的文件将不被前端控制器拦截 -->
```

```
    <mvc:resources location="/js/" mapping="/js/**"></mvc:resources>
    <mvc:resources location="/css/" mapping="/css/**"></mvc:resources>
    <mvc:resources location="/images/" mapping="/images/**"></mvc:resources>
    <!-- 定义视图解析器 -->
    <bean id="viewResoler"
     class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     <!-- 设置前缀 -->
     <property name="prefix" value="/WEB-INF/jsp/" />
     <!-- 设置后缀 -->
     <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

上述代码除了配置需要扫描的包、注解驱动和视图解析器外，还增加了加载属性文件和访问静态资源的配置。

除以上配置文件外，还需要在项目的/WebContent/WEB-INF 目录下编写 web.xml 文件，如文件 16.5 所示。

文件 16.5　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">
    <!-- 配置加载 Spring 文件的监听器 -->
    <context-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
     <listener-class>
         org.springframework.web.context.ContextLoaderListener
     </listener-class>
    </listener>
    <!-- 编码过滤器 -->
    <filter>
     <filter-name>encoding</filter-name>
     <filter-class>org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
     <init-param>
         <param-name>encoding</param-name>
         <param-value>UTF-8</param-value>
     </init-param>
    </filter>
    <filter-mapping>
```

```
        <filter-name>encoding</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
    <!-- 配置 Spring MVC 前端控制器 -->
    <servlet>
        <!-- 配置前端过滤器 -->
        <servlet-name>springmvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 初始化时加载配置文件 -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <!-- 配置服务器启动时立即加载 Spring MVC 配置文件 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>*.action</url-pattern>
    </servlet-mapping>
    <!-- 系统默认页面 -->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

在 web.xml 文件中配置了 Spring 的监听器、编码过滤器和 Spring MVC 的前端控制器等信息。

16.4.3 创建项目相关目录（包）和文件，并引入相关文件资源

按照如图 16.2 所示的项目文件组织结构创建项目相关的目录（包）及文件，如相关类和接口的包、JSP 文件对应的文件夹、JSP 文件等，并引入项目开发需要的相关文件资源，如 CSS 样式文件、images 图片文件、JS 文件、标签文件等，为项目开发做好准备工作。在后续项目开发过程中，根据需要可以创建其他文件或引入其他资源文件。

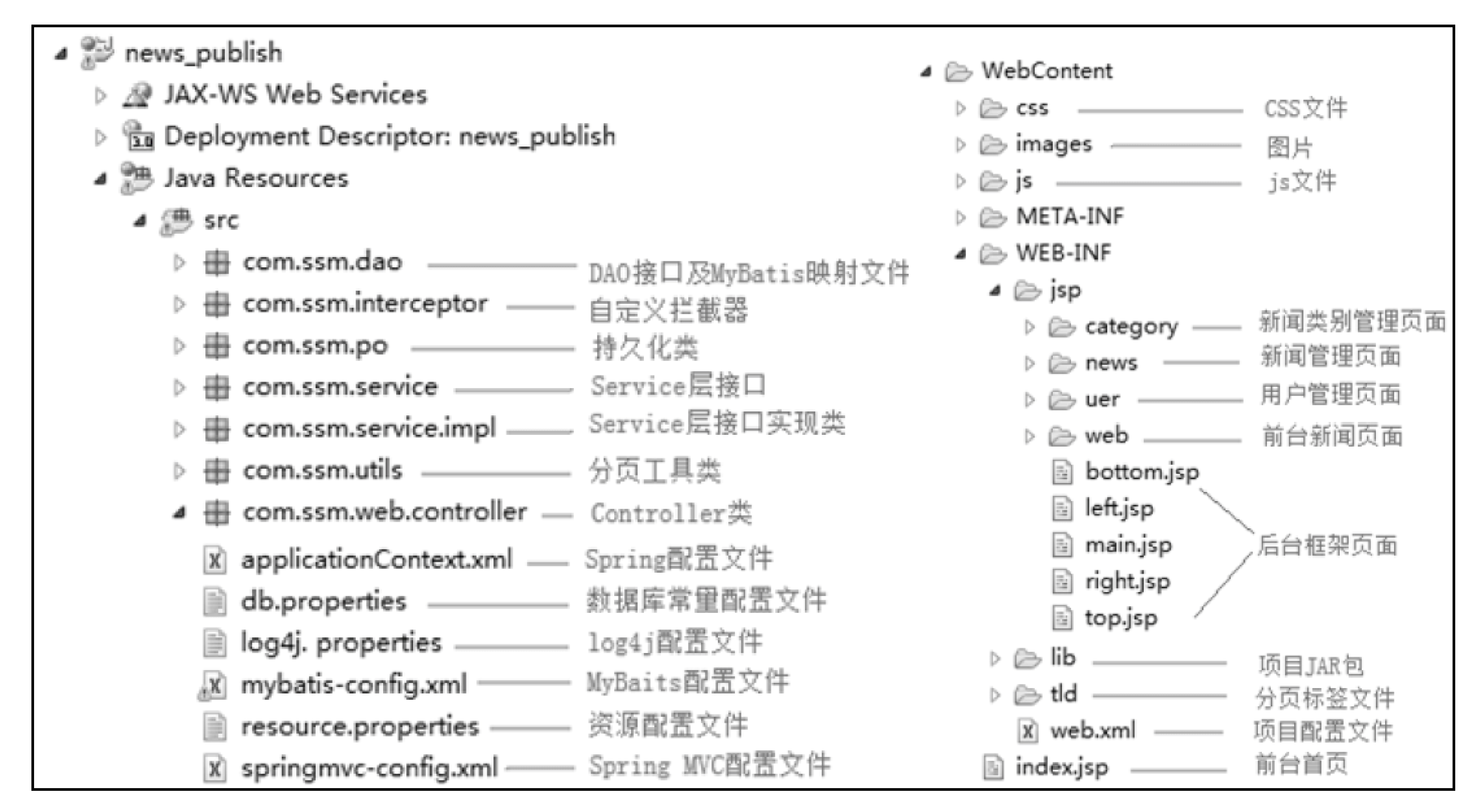


图 16.2　项目文件组织结构

16.5　用户管理模块

用户管理模块涉及用户添加、修改、查询、删除、设置角色和登录及退出等功能。接下来对这些功能进行具体实现。

16.5.1　创建持久化类

用户管理模块持久化类有角色类 Role 和用户类 User，具体如文件 16.6 和文件 16.7 所示。

文件 16.6　Role.java

```
package com.ssm.po;
import java.util.List;
//角色类
public class Role {
    private Integer roleId;     //角色 id
    private String roleName;   //角色名称
    private List<User> userList;  //对应角色的用户列表
    public Integer getRoleId() {
     return roleId;
    }
    public void setRoleId(Integer roleId) {
     this.roleId = roleId;
    }
```

```
    public String getRoleName() {
     return roleName;
    }
    public void setRoleName(String roleName) {
     this.roleName = roleName;
    }
    public List<User> getUserList() {
     return userList;
    }
    public void setUserList(List<User> userList) {
     this.userList = userList;
    }
    @Override
    public String toString() {
     return "Role [roleId=" + roleId + ", roleName=" + roleName + ", userList=" + userList
    + "]";
    }
}
```

文件 16.7 User.java

```
package com.ssm.po;
import java.util.Date;
//用户类
public class User {
    private Integer userId;     //用户 id
    private String userName;  //用户姓名
    private String loginName;  //用户登录名
    private String password;   //登录密码
    private String tel;         //联系电话号码
    private Date registerTime;  //注册或修改用户时间
    private String status;      //用户状态（'1'：未启用；'2'：已启用；'3'：被禁用）
    private Integer roleId;     //用户对应的角色 id
    private String roleName;   //角色名称（为了方便列表页显示角色名，增加此属性）
    public Integer getUserId() {
     return userId;
    }
    public void setUserId(Integer userId) {
     this.userId = userId;
    }
    public String getUserName() {
     return userName;
    }
    public void setUserName(String userName) {
     this.userName = userName;
    }
    public String getLoginName() {
```

```java
        return loginName;
    }
    public void setLoginName(String loginName) {
        this.loginName = loginName;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getTel() {
        return tel;
    }
    public void setTel(String tel) {
        this.tel = tel;
    }
    public Date getRegisterTime() {
        return registerTime;
    }
    public void setRegisterTime(Date registerTime) {
        this.registerTime = registerTime;
    }
    public String getStatus() {
        return status;
    }
    public void setStatus(String status) {
        this.status = status;
    }
    public Integer getRoleId() {
        return roleId;
    }
    public void setRoleId(Integer roleId) {
        this.roleId = roleId;
    }
    public String getRoleName() {
        return roleName;
    }
    public void setRoleName(String roleName) {
        this.roleName = roleName;
    }
    @Override
    public String toString() {
        return "User [userId=" + userId + ", userName=" + userName + ", loginName=" +
              loginName + ", password=" + password + ", tel=" + tel + ", registerTime=" +
             registerTime + ", status=" + status + ", roleId="                + roleId +
```

```
            ", roleName=" + roleName + "]";
    }
}
```

16.5.2 实现 DAO

步骤01 创建 DAO 层接口。在 src 目录下的 com.ssm.dao 包中创建一个角色接口 RoleDao 和一个用户接口 UserDao，并在接口中编写增、删、改、查等方法，如文件 16.8 和文件 16.9 所示。

文件 16.8 RoleDao.java

```
package com.ssm.dao;
import java.util.List;
import org.apache.ibatis.annotations.Param;
import com.ssm.po.Role;
public interface RoleDao {
  //获取所有角色信息（角色列表）
    public List<Role> selectRoleList();
}
```

文件 16.9 UserDao.java

```
package com.ssm.dao;
import java.util.List;
import org.apache.ibatis.annotations.Param;
import com.ssm.po.User;
/*
 * 用户 DAO 层接口
 */
public interface UserDao {
    //查询所有用户
    public List<User> selectUserList(@Param("keywords") String keywords,
                         @Param("userListRoleId") Integer userListRoleId);
    //通过账号和密码查询用户
    public User findUser(@Param("loginName") String loginName,
                @Param("password") String password);
    //通过用户 id 查询用户
public User getUserByUserId(Integer userId);
    //通过用户登录名查询用户（用于判断用户名是否已存在）
    public User getUserByLoginName(String loginName);
//添加用户
    public int addUser(User user);
//更新用户
    public int updateUser(User user);
//删除用户
    public int delUser(Integer userId);
//设置用户状态（'1'：未启用；'2'：已启用；'3'：被禁用）
```

```
    public int setUserStatus(User user);
}
```

步骤02 创建映射文件。在 src 目录下的 com.ssm.dao 包中创建 MyBatis 映射文件 RoleDao.xml 和 UserDao.xml，并在映射文件中编写增、删、改、查等方法的执行语句，如文件 16.10 和文件 16.11 所示。

文件 16.10　RoleDao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.dao.RoleDao">
    <!--查询角色集合列表 -->
    <select id="selectRoleList" resultType="Role">
     select roleId,roleName from t_role
    </select>
    </resultMap>
</mapper>
```

文件 16.11　UserDao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.dao.UserDao">
    <!--查询所有用户集合 where 语句 -->
    <sql id="selectUserListWhere">
      <where>
          u.roleId=r.roleId
        <if test="keywords!=null and keywords!=''">
           and (u.username like CONCAT('%',#{keywords},'%') or
             u.loginName like CONCAT('%',#{keywords},'%'))
        </if>
        <if test="userListRoleId!=null and userListRoleId!=''">
           and (u.roleId=#{userListRoleId})
        </if>
      </where>
    </sql>
    <!--查询所有用户集合列表 -->
    <select id="selectUserList" parameterType="String" resultType="User">
       select u.*,r.roleName from t_user as u,t_role as r
     <include refid="selectUserListWhere" />
     order by registerTime desc
    </select>
    <!--通过账号和密码查询用户 -->
    <select id="findUser" parameterType="String" resultType="User">
     select * from t_user where loginName=#{loginName} and password=#{password}
```

```
            limit 0,1
    </select>
    <!--通过 userId 查询用户 -->
    <select id="getUserByUserId" parameterType="Integer" resultType="User">
     select * from t_user where userId=#{userId}
    </select>
    <!--通过登录账号查询用户 -->
    <select id="getUserByLoginName" parameterType="String" resultType="User">
     select * from t_user where loginName=#{loginName} limit 0,1
    </select>
    <!--添加用户 -->
    <insert id="addUser" parameterType="User">
     insert into t_user(
         userName,
         loginName,
         password,
         tel,
         registerTime,
         status,
         roleId
     )
     values(
         #{userName},
         #{loginName},
         #{password},
         #{tel},
         #{registerTime},
         #{status},
         #{roleId}
     )
    </insert>
    <!-- 更新用户 -->
    <update id="updateUser" parameterType="User">
     update t_user
     <set>
         registerTime=#{registerTime},
         status=#{status},
     <if test="userName!=null and userName!=''">
         userName=#{userName},
     </if>
     <if test="password!=null and password!=''">
         password=#{password},
     </if>
     <if test="tel!=null and tel!=''">
         tel=#{tel},
     </if>
```

```
        <if test="roleId!=null and roleId!=''">
            roleId=#{roleId},
        </if>
        </set>
        where userId=#{userId}
       </update>
       <!--删除用户 -->
       <delete id="delUser" parameterType="Integer">
        delete from t_user where userId=#{userId}
       </delete>
       <!--设置用户状态（status '1':未启用; '2': 已启用; '3': 被禁用）-->
       <update id="setUserStatus" parameterType="User">
        update t_user set status=#{status} where userId=#{userId}
       </update>
</mapper>
```

文件 16.11 中的代码通过映射查询、增加、修改和删除等语句来实现用户表中的操作。

16.5.3　实现 Service

步骤01 创建角色和用户的 Service 层接口。在 src 目录下创建一个 com.ssm.service 包，在包中创建 RoleService 接口和 UserService 接口，并在该接口中编写操作用户的相关方法，如文件 16.12 和文件 16.13 所示。

文件 16.12　RoleService.java

```
package com.ssm.service;
import java.util.List;
import com.ssm.po.Role;
/*
 * 角色 Service 层接口
 */
public interface RoleService {
    public List<Role> findRoleList();
}
```

文件 16.13　UserService.java

```
package com.ssm.service;
import java.util.List;
import com.ssm.po.User;
/*
 * 用户 Service 层接口
 */
public interface UserService {
    public List<User> findUserList(String keywords, Integer userListRoleId);
    public User findUser(String loginName,String password);
    public User getUserByUserId(Integer userId);
```

```
    public User getUserByLoginName(String loginName);
    public int editUser(User user);
    public int addUser(User user);
    public int delUser(Integer userId);
    public int setUserStatus(User user);
}
```

步骤02 创建角色和用户 Service 层接口的实现类。在 src 目录下创建一个 com.ssm.service.impl 包，并在包中创建 RoleService 接口的实现类 RoleServiceImpl 和 UserService 接口的实现类 UserServiceImpl，在类中编辑并实现接口中的方法，如文件 16.14 和文件 16.15 所示。

文件 16.14 RoleServiceImpl.java

```
package com.ssm.service.impl;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.ssm.dao.RoleDao;
import com.ssm.po.Role;
import com.ssm.service.RoleService;
@Service("roleService")
/*
 * 角色 Service 接口实现类
 */
public class RoleServiceImpl implements RoleService {
    //注入 RoleDao
    @Autowired
    private RoleDao roleDao;
    @Override
    public List<Role> findRoleList() {
     List<Role> roleList=roleDao.selectRoleList();
     return roleList;
    }
}
```

文件 16.15 UserServiceImpl.java

```
package com.ssm.service.impl;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.ssm.dao.UserDao;
import com.ssm.po.User;
import com.ssm.service.UserService;
@Service("userService")
/*
 * 用户 Service 接口实现类
 */
```

```java
public class UserServiceImpl implements UserService{
    //注入 UserDao
    @Autowired
    private UserDao userDao;
    @Override
    public List<User> findUserList(String keywords, Integer userListRoleId) {
     List<User> userList=this.userDao.selectUserList(keywords,userListRoleId);
     return userList;
    }
    @Override
    public User findUser(String loginName, String password) {
     User user=this.userDao.findUser(loginName, password);
     return user;
    }
    @Override
    public User getUserByUserId(Integer userId) {
     return this.userDao.getUserByUserId(userId);
    }
    @Override
    public User getUserByLoginName(String loginName) {
     return this.userDao.getUserByLoginName(loginName);
    }
    @Override
    public int addUser(User user) {
     return userDao.addUser(user);
    }
    @Override
    public int editUser(User user) {
     return this.userDao.updateUser(user);
    }
    @Override
    public int delUser(Integer userId) {
     return userDao.delUser(userId);
    }
    @Override
    public int setUserStatus(User user) {
     return userDao.setUserStatus(user);
    }
}
```

16.5.4　实现 Controller

在 src 目录下创建一个 com.ssm.web.controller 包，在包中创建用户控制器类 UserController，编辑后的代码如文件 16.16 所示。

文件 16.16 UserController.java

```
package com.ssm.web.controller;
import java.util.Date;
import java.util.List;
import javax.servlet.http.HttpSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import com.ssm.po.Role;
import com.ssm.po.User;
import com.ssm.service.RoleService;
import com.ssm.service.UserService;
/*
 * 用户控制类
 */
@Controller
public class UserController {
    //依赖注入
    @Autowired
    private UserService userService;
    @Autowired
    private RoleService roleService;
    //查询所有状态的用户集合（用户列表）
    @RequestMapping(value="/findUserList.action")
    public String findUserList(String keywords,Integer userListRoleId,Model model){
     // 获取角色列表
     List<Role> roleList = roleService.findRoleList();
     model.addAttribute("roleList", roleList);
     // 获取用户列表
     List<User> userList=userService.findUserList(keywords,userListRoleId);
     model.addAttribute("userList", userList);
     model.addAttribute("keywords", keywords);
     model.addAttribute("userListRoleId", userListRoleId);
     return "user/user_list";
    }
    //转向添加用户
    @RequestMapping(value="/toAddUser.action")
    public String toAddUser(Model model){
     //获取角色列表，用于添加用户页面中的用户角色下拉列表
     List<Role> roleList=roleService.findRoleList();
     model.addAttribute("roleList", roleList);
     return "user/add_user";
```

```
    }
    //判断登录账号是否已存在
    @RequestMapping(value = "/checkLoginName.action")
    @ResponseBody
    public User checkLoginName(@RequestBody User user, Model model) {
     User checkUser = userService.getUserByLoginName(user.getLoginName());
     if (checkUser!=null) {
         //登录账号已存在
         return checkUser;
     }else{
         checkUser=new User();
         checkUser.setUserId(0);
         return checkUser;
     }
    }
    //添加用户
    @RequestMapping(value = "/addUser.action", method = RequestMethod.POST)
    public String addUser(User user, Model model) {
     // 获取角色列表
     List<Role> roleList = roleService.findRoleList();
     model.addAttribute("roleList", roleList);
     model.addAttribute("user", user);
     //检查登录账号是否已存在
     User checkUser = userService.getUserByLoginName(user.getLoginName());
     if (checkUser!=null) {
         // 登录账号已存在，重新转回添加用户页面
         model.addAttribute("checkUserLoginNameMsg", "登录账号已存在，请重新输入");
         return "user/add_user";
     }else{
         // 登录账号可用
         Date date = new Date();
         user.setRegisterTime(date);
         //默认设置用户为启用状态"2"
         user.setStatus("2");
         //调用 UserService 实例中的添加用户方法
         int rows = userService.addUser(user);
         if (rows > 0) {
             // 添加成功，转向用户列表页面
             return "redirect:findUserList.action";
         } else {
             // 添加失败，重新转回添加用户页面
             return "user/add_user";
         }
     }
    }
    // 转向修改用户页面
```

```
    @RequestMapping(value = "/toEditUser.action")
    public String toEditUser(Integer userId, Model model) {
        //通过 userId 获取用户
        User user = userService.getUserByUserId(userId);
        if (user != null) {
            model.addAttribute("user", user);
            // 获取角色列表
            List<Role> roleList = roleService.findRoleList();
            model.addAttribute("roleList", roleList);
            return "user/edit_user";
        }else{
            return "redirect:findUserList.action";
        }
    }
    //修改用户
    @RequestMapping(value = "/editUser.action", method = RequestMethod.POST)
    public String editUser(User user, Model model) {
        // 获取角色列表
        Date date = new Date();
        user.setRegisterTime(date);
        // 默认设置用户为启用状态"2"
        user.setStatus("2");
        int rows = userService.editUser(user);
        if (rows > 0) {
            // 添加成功，转向用户列表页面
            return "redirect:findUserList.action";
        } else {
            List<Role> roleList = roleService.findRoleList();
            model.addAttribute("roleList", roleList);
            model.addAttribute("user", user);
            // 修改失败，转回修改用户页面
            return "user/edit_user";
        }
    }
    //删除用户（前台页面中通过 jQuery Ajax 方式调用此方法）
    @RequestMapping(value = "/delUser.action")
    @ResponseBody
    public User delUser(@RequestBody User user, Model model) {
        int rows = userService.delUser(user.getUserId());
        if (rows>0) {
            return user;
        }else{
            //此处设置 userId 为 0，只是作为操作失败的标记用
            user.setUserId(0);
            return user;
        }
```

```
    }
    //禁用用户（更新 status 字段值为'3'，前台页面中通过 jQuery Ajax 方式调用此方法）
    @RequestMapping(value = "/disableUser.action")
    @ResponseBody
    public User disableUser(@RequestBody User user, Model model) {
     int rows = userService.setUserStatus(user);
     if (rows>0) {
         return user;
     }else{
         //此处设置 userId 为 0，只是作为操作失败的标记用
         user.setUserId(0);
         return user;
     }
    }
    //启用用户（更新 status 字段值为'2'，前台页面中通过 jQuery Ajax 方式调用此方法）
    @RequestMapping(value = "/enableUser.action")
    @ResponseBody
    public User enableUser(@RequestBody User user, Model model) {
     int rows = userService.setUserStatus(user);
     if (rows>0) {
         return user;
     }else{
         //此处设置 userId 为 0，只是作为操作失败的标记用
         user.setUserId(0);
         return user;
     }
    }
    //用户登录
    @RequestMapping(value="/login.action",method=RequestMethod.POST)
    public String login(String loginName,String password,Model model,HttpSession session){
     //通过用户名和密码查询用户
     User user=userService.findUser(loginName, password);
     if(user!=null){
         if (user.getStatus().equals("2")) {
             //用户被启用时，允许登录到后台
             session.setAttribute("login_user", user);
             return "main";
         } else {
             //账号未启用或被禁用时，不允许登录到后台
             model.addAttribute("msg", "账号未启用或被禁用，请联系管理员！");
             return "login";
         }
     }
     //账号或密码错误时，不允许登录到后台
     model.addAttribute("msg","账号或密码错误，请重新登录！");
     return "login";
```

```
        }
        //退出登录
        @RequestMapping(value="/logout.action")
        public String logout(HttpSession session){
         //清空 session
         session.invalidate();
         return "login";
        }
}
```

文件 16.16 中，首先通过@Autowired 注解将 RoleService 对象和 UserService 对象注入本类中，然后创建对用户进行增、删、改、查、登录、退出等方法。

注　意

UserController 类的一些方法中涉及一些业务逻辑，请读者细心阅读，并将 UserController 类与系统中的用户管理相关页面联系起来推敲。

（1）UserController 类中每个方法前通过注解@RequestMapping 设置了方法的访问路径，例如@RequestMapping(value = "/addUser.action", method = RequestMethod.POST)。

（2）addUser()方法用于添加用户。添加用户前，先调用 toAddUser()方法获取角色集合列表传递到添加用户页面，并通过 c 标签循环添加到角色选择的下拉列表框的选项中。在添加用户页面输入用户信息后，单击"添加"按钮，调用 addUser()方法，addUser()方法先调用 UserService 接口实例的 getUserByLoginName()方法判断输入的登录账号是否已经存在（即不可再用）。如果已存在，就返回添加用户页面，并给出提示信息；如果不存在，就继续添加用户，添加成功的话，转向用户列表页面，添加不成功则返回添加用户页面，让用户重新添加用户。

（3）login()方法用于用户登录。由于在用户登录时，表单会以 POST 方式提交，因此将 RequestMapping 注解的 method 属性值设置为 RequestMethod.POST。在 login()方法中，首先通过页面中传递过来的账号和密码查询用户，然后通过 if 语句判断是否存在该用户。如果不存在，就提示错误信息，并返回登录页面；如果存在，但用户账号未启用或被禁用，就提示相关信息，并返回登录页面；如果存在且已启用，就将用户信息存储到 session 中，并跳转到系统后台主页面。

（4）删除用户 delUser()方法、禁用用户 disableUser()方法和启用用户 enableUser()方法，系统页面采用 jQuery Ajax 方式进行调用。相关交互方式在第 13 章 JSON 数据交互中进行了介绍，在此不再赘述，请读者参考第 13 章中的相关内容。

（5）系统页面与 UserController 类中方法交互过程的参数传递在前面章节中有相关介绍，请读者参考前面相关章节中的内容。

16.5.5　实现页面功能

步骤01 在项目 WebContent 目录下创建 login.jsp 页面文件。login.jsp 主要实现了用户登录到后台的功能，其实现代码如文件 16.17 所示。

文件 16.17　login.jsp

```
<!-- 此处省略部分代码，具体请参见本章案例代码 -->
<div class="login">
    <div class="header">
        <div class="switch">
            <a href="#" class="switch_login" id="switch_login">后台登录</a>
                <div class="switch_bottom" id="switch_bottom"
                   style="position: absolute;width:70px;left:0px;">
                </div>
            </div>
        </div>
        <!--登录-->
        <div class="qlogin" id="qlogin">
            <div class="web_login" align="center">
                <form action="${pageContext.request.contextPath}/login.action"
                         method="post"  onsubmit="return checkValue()">
                <font style="color:#ff0000;font-size:16px;">${msg }</font>
                <div style="height:10px;"></div>
                <div class="uname" class="input-group">
                    <span class="input-group-addon">
            <img src="${pageContext.request.contextPath}/images/username.png" />
                    </span>
                    <input type="text" name="loginName" id="loginName"
                       class="form-control" placeholder="请输入登录账号"/>
                </div>
                <div class="upwd" class="input-group">
                    <span class="input-group-addon">
            <img src="${pageContext.request.contextPath}/images/password.png" />
                    </span>
                <input type="password" name="password" id="password"
                        class="form-control" placeholder="密码"/>
                </div>
                <div style="padding-left: 100px; margin-top: 20px;">
                    <button class="btn btn-primary btn-lg" style="width:150px">
                     登录
                   </button>
                </div>
                </form>
            </div>
        </div>
    </div>
        <script>
        function checkValue() {
            var str = document.getElementById("loginName").value;
            if (str.length <1) {
                alert("请输入账号");
```

```
                document.getElementById("loginName").focus();
                return false;
            }
            str = document.getElementById("password").value;
            if (str.length <1) {
                alert("请输入密码");
                document.getElementById("password").focus();
                return false;
            }
            return true;
        }
    </script>
<!-- 此处省略部分代码，具体请参见本章案例代码 -->
```

在文件 16.17 中，核心代码是用户登录操作的 form 表单，该表单在提交时会通过 checkValue()方法检查账户或密码是否为空，如果为空，就通过 alert()消息框提示用户；如果账号和密码都已填写，就将表单提交到以“/login.action”结尾的请求中。

步骤02 在 WEB-INF 目录下创建 jsp 文件夹（与前面所讲的配置文件 springmvc-config.xml 中的前缀配置相对应），并在 jsp 文件下创建 main.jsp、top.jsp、left.jsp、right.jsp 页面。这几个页面主要实现后台框架页面功能，以及根据登录后保存在 session 中的用户信息的角色显示与其权限相适应的操作界面和菜单选项的功能，具体代码请读者参考本章案例代码。

步骤03 在 WEB-INF/jsp 目录下创建 user 文件夹，并在该文件夹下创建用户列表页面 user_list.jsp、添加用户页面 add_user.jsp 和更新用户页面 edit_user.jsp，具体实现代码如文件 16.18、文件 16.19 和文件 16.20 所示。

文件 16.18 user_list.jsp

```
<!-- 此处省略部分代码，具体请参见本章案例代码 -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"  prefix="fmt"%>
<!-- 此处省略部分代码 -->
<link href="${pageContext.request.contextPath}/css/style.css"
    rel="stylesheet" type="text/css" />
 <script type="text/javascript"
    src="${pageContext.request.contextPath}/js/jquery-1.11.3.min.js"></script>
 </head>
<body>
    <!-- 此处省略部分代码 -->
    <div class="tools">
    <ul class="toolbar">
       <li class="click">
          <a href="${pageContext.request.contextPath}/toAddUser.action">
             <span><img src="images/t01.png" /></span>添加用户
          </a>
         </li>
```

```
        </ul>
        </div>
        <form action="${pageContext.request.contextPath}/findUserList.action"
             id="userListForm" name="ff" method="post">
           <ul class="seachform">
               <li>
                 <label>    搜索关键词:</label>
                  <input name="keywords" value="${keywords}"
                       placeholder="这里输入姓名或登录账号" type="text"
                      style="width: 400px;" class="scinput" />
              </li>
               <li>
                <label style="width: 60px;">角色: </label>
                <select name="userListRoleId" id="userListRoleId" class="dfinput">
                        <option value="">--请选择--</option>
                        <c:forEach items="${roleList}" var="r">
                            <option value="${r.roleId}"
                   <c:if test="${r.roleId eq userListRoleId }">selected="selected"</c:if> >
                           ${r.roleName }
                          </option>
                        </c:forEach>
                  </select>
              </li>
               <li><input name="" type="submit" class="scbtn" value="查询" /></li>
           </ul>
       </form>
       <table class="tablelist">
           <thead>
           <tr>
             <th style="width:100px;text-align:center;">用户姓名</th>
             <th style="width:100px;text-align:center;">登录账号</th>
             <th style="width:100px;text-align:center;">联系电话</th>
             <th style="width:100px;text-align:center;">注册/修改时间</th>
             <th style="width:120px;text-align:center;">用户角色</th>
             <th style="width:120px;text-align:center;">审核状态</th>
             <th style="width:200px;text-align:center;">操作</th>
             </tr>
             </thead>
               <tbody>
               <c:if test="${!empty userList}">
                   <c:forEach items="${userList}" var="user">
                       <tr>
                           <td align="center">${user.userName}</td>
                           <td align="center">${user.loginName}</td>
                           <td align="center">${user.tel}</td>
                           <td align="center">
           <fmt:formatDate value="${user.registerTime}" pattern="yyyy-MM-dd HH:mm:ss"/>
```

```
                        </td>
                          <td align="center">${user.roleName}</td>
                          <td align="center">
                          <c:if test="${user.status=='2'}" var="flag">
                              已启用
                         </c:if>
                        <c:if test="${not flag}">
                             未启用或被禁用
                         </c:if>
                          </td>
                         <td align="center">
    <a href="${pageContext.request.contextPath}/toEditUser.action?userId=${user.userId}">
    修改</a>
                     <c:if test="${user.loginName!='admin'}">
                          <c:if test="${user.status=='2'}" var="status_flag">
                                     | 
                        <a href='#' onclick="disableUser(${user.userId})">禁用</a>
                         </c:if>
                         <c:if test="${not status_flag}">
                           | 
                         <a href='#' onclick="enableUser(${user.userId})">启用</a>
                         </c:if>
                                | 
                          <a href='#' onclick="del(${user.userId})">删除</a>
                          </c:if>
                     </td>
                     </tr>
                </c:forEach>
           </c:if>
        <c:if test="${empty userList}">
           <div>
                <tr>
                <td colspan="7" align="center">
                暂无用户
                </td>
                </tr>
           </div>
       </c:if>
       </tbody>
   </table>
     <script type="text/javascript">
    <!-- 禁用用户 -->
   function disableUser(userId){
       $.ajax({
           url:"${pageContext.request.contextPath }/disableUser.action",
           type:"post",
```

```
                //data 表示发送的数据
                data:JSON.stringify({userId:userId,status:3}),
                // 定义发送请求的数据格式为 JSON 字符串
                contentType:"application/json;charset=UTF-8",
                //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
                dataType:"json",
                //成功响应的结果
                success:function(data){
                    if(data!=null){
                        if(data.userId>0){
                            alert("禁用成功！");
                            window.location.reload();
                        }else{
                            alert("禁用失败！");
                            window.location.reload();
                        }
                    }
                }
                });
        }
        <!-- 启用用户 -->
        function enableUser(userId){
            $.ajax({
                url:"${pageContext.request.contextPath }/enableUser.action",
                type:"post",
                //data 表示发送的数据
                data:JSON.stringify({userId:userId,status:2}),
                // 定义发送请求的数据格式为 JSON 字符串
                contentType:"application/json;charset=UTF-8",
                //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
                dataType:"json",
                //成功响应的结果
                success:function(data){
                    if(data!=null){
                        if(data.userId>0){
                            alert("启用成功！");
                            window.location.reload();
                        }else{
                            alert("启用失败！");
                            window.location.reload();
                        }
                    }
                }
                });
        }
        <!-- 删除用户 -->
```

```
        function del(userId){
            if(window.confirm("您确定要删除吗？"))
            {
                $.ajax({
                    url:"${pageContext.request.contextPath }/delUser.action",
                    type:"post",
                    //data 表示发送的数据
                    data:JSON.stringify({userId:userId}),
                    // 定义发送请求的数据格式为 JSON 字符串
                    contentType:"application/json;charset=UTF-8",
                    //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
                    dataType:"json",
                    //成功响应的结果
                    success:function(data){
                        if(data!=null){
                            if(data.userId>0){
                                alert("删除成功！");
                                window.location.reload();
                            }else{
                                alert("删除失败！");
                                window.location.reload();
                            }
                        }
                    }
                });
            }
        }
       </script>
      </div>
      <script type="text/javascript">
            $('.tablelist tbody tr:odd').addClass('odd');
         </script>
</body>
</html>
```

用户列表页面 user_list.jsp 页面文件中主要用来展示用户列表，通过访问 UserController 类的 findUserList()可以跳转到 user_list.jsp 页面并在页面中迭代显示用户列表信息，并在此基础上可以通过链接对用户进行修改、删除、禁用/启用等相关操作。

文件 16.19 add_user.jsp

```
<!-- 此处省略部分代码 -->
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!-- 此处省略部分代码 -->
      <form action="${pageContext.request.contextPath}/addUser.action" name="ff"
        method="post" onsubmit="return checkValue()">
<!-- 此处省略部分代码 -->
```

```
        登录账号:
         <input id="loginName" name="loginName" value="${user.loginName}" type="text" />
          <font style="color:Red;">${checkUserLoginNameMsg}</font>
        密码:
          <input id="password" name="password" type="password" />
            用户姓名:
           <input id="userName" name="userName"value="${user.userName}" type="text"/>
            电话号码:
        <input id="tel" name="tel"  value="${user.tel}" type="text" />
        用户角色:
           <select name="roleId" id="roleId">
              <option value="">--请选择--</option>
              <c:forEach items="${roleList}" var="r">
                <option value="${r.roleId}"
                  <c:if test="${r.roleId eq user.roleId }">
                     selected="selected"
                  </c:if>>
                  ${r.roleName }
                </option>
              </c:forEach>
         </select>
       <input name="" type="submit" value="确认添加"/>
          <input name="" type="button" onclick="goback();" value="返回列表"/>
        </form>
        <script>
        function goback(){
         window.location.href="${pageContext.request.contextPath}/findUserList.action";
        }
        function checkValue() {
         var str = document.getElementById("loginName").value;
         if (str.length <1) {
              alert("请输入登录账号");
              document.getElementById("loginName").focus();
              return false;
         }
         str = document.getElementById("password").value;
         if (str.length <1) {
              alert("请输入密码");
              document.getElementById("password").focus();
              return false;
         }
         if (str.length > 0 && str.length<6) {
              alert("密码长度应大于等于 6");
              document.getElementById("password").focus();
              return false;
         }
```

```
        str = document.getElementById("userName").value;
        if (str.length <1) {
            alert("请输入用户姓名");
            document.getElementById("userName").focus();
            return false;
        }
        str = document.getElementById("roleId").value;
        if (str == '') {
            alert("请选择用户角色");
            return false;
        }
        return true;
    }
</script>
<!-- 此处省略部分代码 -->
```

文件 16.20　edit_user.jsp

```
<!-- 此处省略部分代码 -->
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!-- 此处省略部分代码 -->
      <form action="${pageContext.request.contextPath}/addUser.action" name="ff"
        method="post" onsubmit="return checkValue()">
   <input id="userId" name="userId" value="${user.userId}" type="hidden" />
<!-- 此处省略部分代码 -->
    登录账号:
     <input id="loginName" name="loginName" value="${user.loginName}" type="text" />
      <font style="color:Red;">${checkUserLoginNameMsg}</font>
    密码:
      <input id="password" name="password" type="password" />
        用户姓名:
       <input id="userName" name="userName"value="${user.userName}" type="text"/>
        电话号码:
    <input id="tel" name="tel"  value="${user.tel}" type="text" />
    用户角色:
       <select name="roleId" id="roleId" >
          <option value="">--请选择--</option>
          <c:forEach items="${roleList}" var="r">
            <option value="${r.roleId}"
              <c:if test="${r.roleId eq user.roleId }">
                 selected="selected"
              </c:if>>
              ${r.roleName }
            </option>
          </c:forEach>
     </select>
   <input name="" type="submit" value="确认添加"/>
```

```
          <input name="" type="button" onclick="goback();" value="返回列表"/>
        </form>
        <script>
        function goback(){
        window.location.href="${pageContext.request.contextPath}/findUserList.action";
        }
        function checkValue() {
         str = document.getElementById("password").value;
         if (str.length > 0 && str.length<6) {
             alert("密码长度应大于等于 6");
             document.getElementById("password").focus();
             return false;
         }
         return true;
          }
    </script>
    <!-- 此处省略部分代码 -->
```

步骤04 启动项目，测试登录和用户管理。将项目发布到 Tomcat 服务器并启动，访问地址为 http://localhost:8080/news_publish/login.jsp，进入登录页面，即可输入账号和密码登录系统，如图 16.3 所示。登录时，存在 3 种情况：



图 16.3　系统后台登录界面一

- 第 1 种情况是，如果账号或密码错误，则不允许登录。
- 第 2 种情况是，如果账号未启用或被禁用，则不允许登录，分别如图 16.4 和图 16.5 所示。



图 16.4　系统后台登录界面二



图 16.5　系统后台登录界面三

- 第 3 种情况是，若账号和密码正确，则允许登录系统后台，登录后根据用户的角色不同，后面操作界面和功能菜单选项会不同。如图 16.6 所示为管理员后台界面；如图 16.7 所示为信息员后台界面。



图 16.6　管理员系统后台界面



图 16.7　信息员系统后台界面

管理员登录系统后台后，可以对系统用户进行相关操作。如图 16.8、图 16.9、图 16.10 所示的界面中，管理员可以进行查询用户、删除用户、启用/禁用用户、修改用户和添加用户操作。为了保证管理员 admin 不被误删除和禁用，页面代码中通过<c:if>进行了判断和限制操作，即只可以进行修改密码等操作。



图 16.8　查询用户界面

图 16.9　添加用户界面



图 16.10　删除用户界面

从以上测试效果可以看出，已实现了用户登录和用户管理模块的相关功能。

16.6　新闻管理模块

新闻管理模块涉及新闻发布、修改、查询、删除、撤稿等功能。接下来对这些功能进行具体实现。

16.6.1　创建持久化类

用户管理模块持久化类有新闻类别类 Category 和新闻类 News，具体如文件 16.21 和文件 16.22 所示。

文件 16.21　Category.java

```java
package com.ssm.po;
//新闻类别实体类
public class Category {
    private Integer categoryId;     //新闻类别 id
    private String categoryName;   //新闻类别名称
    public Integer getCategoryId() {
```

```
        return categoryId;
    }
    public void setCategoryId(Integer categoryId) {
        this.categoryId = categoryId;
    }
    public String getCategoryName() {
        return categoryName;
    }
    public void setCategoryName(String categoryName) {
        this.categoryName = categoryName;
    }
    @Override
    public String toString() {
        return "Category [categoryId=" + categoryId + ", categoryName=" + categoryName + "]";
    }
}
```

文件 16.22　News.java

```
package com.ssm.po;
import java.util.Date;
//新闻实体类
public class News {
    private Integer newsId;   //新闻 id
    private String title;       //新闻标题（用于新闻列表页）
    private String contentTitle;  //新闻内容页标题（用于新闻内容页）
    private String content;      //新闻内容
    private String contentAbstract;   //内容摘要
    private String keywords;       //新闻关键词
    private String author;          //作者或来源
    private Date publishTime;       //发布时间
    private Integer clicks;          //点击率（量）
    private String publishStatus;     //发布状态（'1'：发布；'2'：撤稿）
    private Integer categoryId;       //新闻类别 id
    private String categoryName;    //新闻类别名称（为了方便新闻列表页显示，特添加此属性）
    private Integer userId;
    public Integer getNewsId() {
        return newsId;
    }
    public void setNewsId(Integer newsId) {
        this.newsId = newsId;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
```

```
    }
    public String getContentTitle() {
     return contentTitle;
    }
    public void setContentTitle(String contentTitle) {
     this.contentTitle = contentTitle;
    }
    public String getContent() {
     return content;
    }
    public void setContent(String content) {
     this.content = content;
    }
    public String getContentAbstract() {
     return contentAbstract;
    }
    public void setContentAbstract(String contentAbstract) {
     this.contentAbstract = contentAbstract;
    }
    public String getKeywords() {
     return keywords;
    }
    public void setKeywords(String keywords) {
     this.keywords = keywords;
    }
    public String getAuthor() {
     return author;
    }
    public void setAuthor(String author) {
     this.author = author;
    }
    public Date getPublishTime() {
     return publishTime;
    }
    public void setPublishTime(Date publishTime) {
     this.publishTime = publishTime;
    }
    public Integer getClicks() {
     return clicks;
    }
    public void setClicks(Integer clicks) {
     this.clicks = clicks;
    }
    public String getPublishStatus() {
     return publishStatus;
    }
```

```
    public void setPublishStatus(String publishStatus) {
     this.publishStatus = publishStatus;
    }
    public Integer getCategoryId() {
     return categoryId;
    }
    public void setCategoryId(Integer categoryId) {
     this.categoryId = categoryId;
    }
    public String getCategoryName() {
     return categoryName;
    }
    public void setCategoryName(String categoryName) {
     this.categoryName = categoryName;
    }
    public Integer getUserId() {
     return userId;
    }
    public void setUserId(Integer userId) {
     this.userId = userId;
    }
    @Override
    public String toString() {
     return "News [newsId=" + newsId + ", title="+title+",contentTitle="+contentTitle +
          ", content=" + content+ ", contentAbstract=" + contentAbstract + ", keywords="
          + keywords + ", author=" + author+ ", publishTime=" + publishTime +",clicks=" +
          clicks + ", publishStatus=" + publishStatus+ ", categoryId=" + categoryId +
          ", categoryName=" + categoryName + ", userId=" + userId + "]";
    }
}
```

后台新闻管理列表页需要进行分页管理，在项目 src 目录下创建包 com.ssm.utils，并定义分页类 PageBean，如文件 16.23 所示。

文件 16.23 PageBean.java

```
package com.ssm.utils;
import java.util.List;
public class PageBean<T> {
    private int currentPage;   //当前页码
    private int pageSize;      //每页显示新闻条数
    private int count;      //新闻数量
    private int totalPage;     //总页数
    private List<T> list;      //当前页的新闻集合列表
    public int getCurrentPage() {
     return currentPage;
    }
    public void setCurrentPage(int currentPage) {
```

```
        this.currentPage = currentPage;
        }
        public int getPageSize() {
        return pageSize;
        }
        public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
        }
        public int getCount() {
        return count;
        }
        public void setCount(int count) {
        this.count = count;
        }
        public int getTotalPage() {
        return totalPage;
        }
        public void setTotalPage(int totalPage) {
        this.totalPage = totalPage;
        }
        public List<T> getList() {
        return list;
        }
        public void setList(List<T> list) {
        this.list = list;
        }
}
```

16.6.2　实现 DAO

步骤01 创建 DAO 层接口。在 src 目录下的 com.ssm.dao 包中创建一个角色接口 CategoryDao 和一个用户接口 NewsDao，并在接口中编写增、删、改、查等方法，如文件 16.24 和文件 16.25 所示。

文件 16.24　CategoryDao.java

```
package com.ssm.dao;
import java.util.List;
import com.ssm.po.Category;
/*
 * 新闻类别 DAO 层接口
 */
public interface CategoryDao {
    //查询所有新闻类别
    public List<Category> selectCategoryList();
    //根据新闻类别 ID 查询新闻类别
```

```
    public Category getCategoryById(Integer categoryId);
}
```

文件 16.25　NewsDao.java

```
package com.ssm.dao;
import java.util.List;
import org.apache.ibatis.annotations.Param;
import com.ssm.po.News;
/*
 * 新闻类别 DAO 层接口
 */
public interface NewsDao {
    //获取当前类别新闻数量
    int getNewsCount(@Param("keywords") String keywords,
              @Param("newsListCategoryId") Integer newsListCategoryId);
    //获取当前类别新闻列表
    List<News> findNewsList(@Param("keywords") String keywords,
                    @Param("newsListCategoryId") Integer newsListCategoryId,
                    @Param("startRows") Integer startRows,
                    @Param("pageSize") Integer pageSize);
//根据新闻 id 获取新闻
    News getNewsByNewsId(Integer newsId);
//添加新闻
    int addNews(News news);
//更新新闻
    int updateNews(News news);
//设置新闻状态（'1'：发布；'2'：撤稿）
    int setNewsPublishStatus(News news);
//删除新闻
    int delNews(Integer newsId);
}
```

步骤02　创建映射文件。在 src 目录下的 com.ssm.dao 包中创建 MyBatis 映射文件 CategoryDao.xml 和 NewsDao.xml，并在映射文件中编写增、删、改、查等方法的执行语句，如文件 16.26 和文件 16.27 所示。

文件 16.26　CategoryDao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.dao.CategoryDao">
    <!--查询新闻类别集合列表 -->
    <select id="selectCategoryList" resultType="Category">
     select * from t_category
    </select>
    <!--通过 categoryId 查询新闻类别 -->
```

```
    <select id="getCategoryById" parameterType="Integer" resultType="Category">
     select * from t_category where categoryId=#{categoryId}
    </select>
</mapper>
```

文件 16.27　NewsDao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.dao.NewsDao">
<!--查询新闻集合 Where -->
    <sql id="selectNewsListWhere">
      <where>
          n.categoryId=c.categoryId
        <if test="keywords!=null and keywords!=''" >
          and (n.title like CONCAT('%',#{keywords},'%') or
               n.keywords like CONCAT('%',#{keywords},'%'))
        </if>
        <if test="newsListCategoryId!=null and newsListCategoryId!=''" >
          and (n.categoryId=#{newsListCategoryId})
        </if>
      </where>
    </sql>
    <!--查询新闻集合 -->
    <select id="findNewsList" parameterType="String" resultType="News">
       select n.*,c.categoryName from t_news as n,t_category as c
     <include refid="selectNewsListWhere" />
     order by publishTime desc
     limit #{startRows},#{pageSize}
    </select>
    <sql id="getNewsCountWhere">
      <where>
        <if test="keywords!=null and keywords!=''" >
          and (n.title like CONCAT('%',#{keywords},'%') or
             n.keywords like CONCAT('%',#{keywords},'%'))
        </if>
        <if test="newsListCategoryId!=null and newsListCategoryId!=''" >
          and (n.categoryId=#{newsListCategoryId})
        </if>
      </where>
    </sql>
    <!--查询新闻数量 -->
    <select id="getNewsCount" parameterType="String" resultType="Integer">
       select count(*) from t_news as n
     <include refid="getNewsCountWhere" />
    </select>
```

```
    <!--通过 newsId 查询新闻 -->
    <select id="getNewsByNewsId" parameterType="Integer" resultType="News">
     select *,categoryName from t_news as n,t_category as c
            where newsId=#{newsId} and n.categoryId=c.categoryId
    </select>
    <!--添加用户 -->
    <insert id="addNews" parameterType="News">
     insert into t_news(
         title,
         contentTitle,
         content,
         contentAbstract,
         keywords,
         author,
         publishTime,
         publishStatus,
         categoryId
     )
     values(
         #{title},
         #{contentTitle},
         #{content},
         #{contentAbstract},
         #{keywords},
         #{author},
         #{publishTime},
         #{publishStatus},
         #{categoryId}
     )
    </insert>
    <!-- 更新用户 -->
    <update id="editNews" parameterType="News">
     update t_news
     <set>
         publishTime=#{publishTime},
         publishStatus=#{publishStatus},
         title=#{title},
         contentTitle=#{contentTitle},
         content=#{content},
         contentAbstract=#{contentAbstract},
         keywords=#{keywords},
         author=#{author},
         categoryId=#{categoryId}
     </set>
     where newsId=#{newsId}
    </update>
```

```
    <!--设置新闻发布状态（status '1':已发布；'2'：被撤稿）-->
    <update id="setNewsPublishStatus" parameterType="News">
     update t_news set publishStatus=#{publishStatus} where newsId=#{newsId}
    </update>
    <!--删除新闻 -->
    <delete id="delNews" parameterType="Integer">
     delete from t_news where newsId=#{newsId}
    </delete>
</mapper>
```

文件 16.27 中的代码通过映射查询、增加、修改和删除等语句来实现新闻表中的操作。

16.6.3　实现 Service

步骤01 创建角色和用户的 Service 层接口。在 src 目录下创建一个 com.ssm.service 包，在包中创建 CategoryService 接口和 NewsService 接口，并在该接口中编写操作用户的相关方法，如文件 16.28 和文件 16.29 所示。

文件 16.28　CategoryService.java

```java
package com.ssm.service;
import java.util.List;
import com.ssm.po.Category;
/*
 * 新闻类别 Service 层接口
 */
public interface CategoryService {
    public List<Category> findCategoryList();
    public Category findCategoryById(Integer categoryId);
}
```

文件 16.29　NewsService.java

```java
package com.ssm.service;
import com.ssm.po.News;
import com.ssm.utils.PageBean;
/*
 * 新闻 Service 层接口
 */
public interface NewsService {
    PageBean<News> findNewsByPage(String keywords, Integer newsListCategoryId,
                                  Integer currentPage, Integer pageSize);
    News getNewsByNewsId(Integer newsId);
    int setNewsPublishStatus(News news);
    int addNews(News news);
    int editNews(News news);
    int delNews(Integer newsId);
}
```

步骤02 创建角色和用户 Service 层接口的实现类。在 src 目录下创建一个 com.ssm.service.impl 包，并在包中创建 CategoryService 接口的实现类 CategoryServiceImpl 和 NewsService 接口的实现类 NewsServiceImpl，在类中编辑并实现接口中的方法，如文件 16.30 和文件 16.31 所示。

文件 16.30 CategoryServiceImpl.java

```
package com.ssm.service.impl;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.ssm.dao.CategoryDao;
import com.ssm.po.Category;
import com.ssm.service.CategoryService;
/*
 * 新闻类别 Service 接口实现类
 */
@Service("categoryService")
public class CategoryServiceImpl implements CategoryService {
    //注入 RoleDao
    @Autowired
    private CategoryDao categoryDao;
    @Override
    public List<Category> findCategoryList() {
     return this.categoryDao.selectCategoryList();
    }
    @Override
    public Category findCategoryById(Integer categoryId) {
     return this.categoryDao.getCategoryById(categoryId);
    }
}
```

文件 16.31 NewsServiceImpl.java

```
package com.ssm.service.impl;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.ssm.dao.NewsDao;
import com.ssm.po.News;
import com.ssm.service.NewsService;
import com.ssm.utils.PageBean;
/*
 * 新闻类别 Service 接口实现类
 */
@Service("newsService")
public class NewsServiceImpl implements NewsService {
    //注入 NewsDao
    @Autowired
```

```
    private NewsDao newsDao;
    @Override
    public PageBean<News> findNewsByPage(String keywords, Integer newsListCategoryId,
                                  Integer currentPage,Integer pageSize) {
     //获取当前类别信息数量
     int count=newsDao.getNewsCount(keywords,newsListCategoryId);
     //求总页数
     int totalPage = (int) Math.ceil(count*1.0/pageSize);
     //获取新闻列表
     List<News> newsList=

    newsDao.findNewsList(keywords,newsListCategoryId,(currentPage-1)*pageSize,pageSiz
    e);
     //创建 PageBean 实例 pb
     PageBean<News> pb = new PageBean<>();
     //向 PageBean 实例 pb 设置值
     pb.setCount(count);
     if(currentPage==0)currentPage=1;
     pb.setCurrentPage(currentPage);
     pb.setList(newsList);
     pb.setPageSize(pageSize);
     pb.setTotalPage(totalPage);
     return pb;
    }
    @Override
    public News getNewsByNewsId(Integer newsId) {
     return newsDao.getNewsByNewsId(newsId);
    }
    @Override
    public int setNewsPublishStatus(News news) {
     return newsDao.setNewsPublishStatus(news);
    }
    @Override
    public int addNews(News news) {
     return newsDao.addNews(news);
    }
    @Override
    public int editNews(News news) {
     return newsDao.updateNews(news);
    }
    @Override
    public int delNews(Integer newsId) {
     return newsDao.delNews(newsId);
    }
}
```

16.6.4 实现 Controller

在 src 目录下创建一个 com.ssm.web.controller 包，在包中创建用户控制器类 NewsController，编辑后的代码如文件 16.32 所示。

文件 16.32 NewsController.java

```
package com.ssm.web.controller;
//此处省略导入包语句
import com.ssm.utils.PageBean;
/*
 * 新闻控制类
 */
@Controller
public class NewsController {
    //依赖注入
    @Autowired
    private NewsService newsService;
    @Autowired
    private CategoryService categoryService;
    //查询新闻分页
    @RequestMapping(value="/findNewsByPage.action")
    public String findNewsByPage(String keywords,Integer newsListCategoryId,
                @RequestParam(defaultValue="1")Integer currentPage,
                @RequestParam(defaultValue="10")Integer pageSize,Model model){
     // 获取角色列表
     List<Category> categoryList = categoryService.findCategoryList();
     model.addAttribute("categoryList", categoryList);
     // 获取用户 PageBean 实例
     PageBean<News> pb=
       newsService.findNewsByPage(keywords,newsListCategoryId,currentPage,pageSize);
     //向 model 添加对象属性，用于页面显示
     model.addAttribute("pb", pb);
     model.addAttribute("keywords", keywords);
     model.addAttribute("newsListCategoryId", newsListCategoryId);
     model.addAttribute("currentPage", currentPage);
     model.addAttribute("pageSize", pageSize);
     return "news/news_list";
    }
    //设置新闻的状态（publishStatus：'1'：发布；'2'：撤稿）
    @RequestMapping(value = "/setNewsPublishStatus.action")
    @ResponseBody
    public News setNewsPublishStatus(@RequestBody News news, Model model) {
     int rows = newsService.setNewsPublishStatus(news);
     if (rows>0) {
         return news;
     }else{
```

```
            news.setNewsId(0);
            return news;
        }
    }
    //转向添加新闻
    @RequestMapping(value = "/toAddNews.action")
    public String toAddNews(Model model) {
        // 获取新闻类别列表
        List<Category> categoryList = categoryService.findCategoryList();
        model.addAttribute("categoryList", categoryList);
        return "news/add_news";
    }
    // 添加新闻
    @RequestMapping(value = "/addNews.action", method = RequestMethod.POST)
    public String addNews(News news, Model model) {
        Date date = new Date();
        news.setPublishTime(date);
        news.setPublishStatus("1");// 默认设置新闻为已发布状态
        int rows = newsService.addNews(news);
        // 添加成功，转向用户列表页面
        if (rows > 0) {
            return "redirect:findNewsByPage.action";
        } else {
            // 获取新闻类别列表
            List<Category> categoryList = categoryService.findCategoryList();
            model.addAttribute("categoryList", categoryList);
            model.addAttribute("news", news);
            // 添加失败，转回添加用户页面
            return "news/add_news";
        }
    }
    // 修改新闻
    @RequestMapping(value = "/editNews.action", method = RequestMethod.POST)
    public String editNews(News news, Model model) {
        Date date = new Date();
        news.setPublishTime(date);
        news.setPublishStatus("1");// 默认设置新闻为已发布状态
        int rows = newsService.editNews(news);
        // 添加成功，转向用户列表页面
        if (rows > 0) {
            return "redirect:findNewsByPage.action";
        } else {
            // 获取新闻类别列表
            List<Category> categoryList = categoryService.findCategoryList();
            model.addAttribute("categoryList", categoryList);
            model.addAttribute("news", news);
```

```
                // 添加失败，转回添加用户页面
                return "news/edit_news";
            }
        }
        // 转向修改新闻页面
        @RequestMapping(value = "/toEditNews.action")
        public String toEditNews(Integer newsId, Model model) {
            News news = newsService.getNewsByNewsId(newsId);
            if (news != null) {
                model.addAttribute("news", news);
                // 获取新闻类别列表
                List<Category> categoryList = categoryService.findCategoryList();
                model.addAttribute("categoryList", categoryList);
            }
            return "news/edit_news";
        }
        //删除新闻
        @RequestMapping(value = "/delNews.action")
        @ResponseBody
        public News delNews(@RequestBody News news, Model model) {
            int rows = newsService.delNews(news.getNewsId());
            if (rows>0) {
                return news;
            }else{
                news.setNewsId(0);
                return news;
            }
        }
        //根据新闻类别 ID 查询新闻分页（用于前台首页）
        @RequestMapping(value = "/index.action")
        public String index(HttpServletRequest request, HttpServletResponse response,
                    String keywords, Integer newsListCategoryId,
                    @RequestParam(defaultValue = "1") Integer currentPage,
                     @RequestParam(defaultValue = "10") Integer pageSize,
                            Model model) throws ServletException, IOException {
            // 获取用户 PageBean 实例（参数新闻类别"今日头条"id 为 1）
            PageBean<News> pb1 =
                        newsService.findNewsByPage(keywords, 1, currentPage, pageSize);
            model.addAttribute("pb1", pb1);
            // 获取用户 PageBean 实例（参数新闻类别"综合资讯"id 为 2）
            PageBean<News> pb2 =
                        newsService.findNewsByPage(keywords, 2, currentPage, pageSize);
            model.addAttribute("pb2", pb2);
            return "../../first";
        }
        //根据新闻类别 ID 查询新闻分页（用于前台新闻列表页）
```

```
    @RequestMapping(value = "/findNewsByCategoryIdPage.action")
    public String findNewsByCategoryIdPage(HttpServletRequest request,
            HttpServletResponse response, String keywords,
            Integer newsListCategoryId,
              @RequestParam(defaultValue = "1") Integer currentPage,
            @RequestParam(defaultValue = "1") Integer pageSize,  Model model){
     // 获取角色列表
     Category category = categoryService.findCategoryById(newsListCategoryId);
     model.addAttribute("category", category);
     // 获取用户 PageBean 实例
     PageBean<News> pb =
    newsService.findNewsByPage(keywords, newsListCategoryId, currentPage, pageSize);
     model.addAttribute("pb", pb);
     model.addAttribute("newsListCategoryId", newsListCategoryId);
     model.addAttribute("currentPage", currentPage);
     model.addAttribute("pageSize", pageSize);
     return "../../list";
    }
    //查询新闻（用于前台新闻内容页）
    @RequestMapping(value = "/findFrontNewsByNewsId.action")
    public String findFrontNewsByNewsId(Integer newsId,Model model) {
     News news = newsService.getNewsByNewsId(newsId);
     if (news != null) {
         model.addAttribute("news", news);
     }
     return "../../detail";
    }
}
```

文件 16.32 中，首先通过@Autowired 注解将 CategoryService 对象和 NewsService 对象注入本类中，然后创建对新闻进行操作的方法。

注　意
NewsController 类与 UserController 类一样，在一些方法中涉及业务逻辑，请读者细心阅读，并将 NewsController 类与系统中的用户管理相关页面联系起来推敲。NewsController 类与 news_list.jsp 页面在交互过程中采用了 PageBean 对新闻进行封装，并实现了分页功能。

16.6.5　实现页面功能

步骤01 在 WEB-INF/jsp 目录下创建 news 文件夹，并在该文件夹下创建新闻列表页面 news_list.jsp、添加新闻页面 add_news.jsp 和更新新闻页面 edit_news.jsp，具体实现代码如文件 16.33、文件 16.34 和文件 16.35 所示。

文件 16.33 news_list.jsp

```
<!-- 此处省略部分代码，具体请参见本章案例代码 -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"  prefix="fmt"%>
<!-- 此处省略部分代码 -->
<link href="${pageContext.request.contextPath}/css/style.css"
    rel="stylesheet" type="text/css" />
<link href="${pageContext.request.contextPath}/css/pagination.css"
    rel="stylesheet" type="text/css" />
 <script type="text/javascript"
    src="${pageContext.request.contextPath}/js/jquery-1.11.3.min.js"></script>
 </head>
<body>
    <!-- 此处省略部分代码 -->
   <div class="tools">
    <ul class="toolbar">
       <li class="click">
          <a href="${pageContext.request.contextPath}/toAddNews.action">
            <span><img src="images/t01.png" /></span>添加新闻
          </a>
         </li>
       </ul>
      </div>
      <form action="${pageContext.request.contextPath}/findNewsByPage.action"
           id="newsListForm" name="ff" method="post">
          <ul class="seachform">
              <li>
                <label>    搜索关键词:</label>
                 <input name="keywords" value="${keywords}"
                     placeholder="这里输入标题或关键词" type="text"
                    style="width: 400px;" class="scinput" />
            </li>
              <li>
               <label style="width: 60px;">新闻类别: </label>
               <select name="newsListCategoryId" id="newsListCategoryId" >
                       <option value="">--请选择--</option>
                    <c:forEach items="${categoryList}" var="c">
                           <option value="${c.categoryId}"
           <c:if test="${c.categoryId eq newsListCategoryId }">selected=
           "selected"</c:if>>
                         ${c.categoryName }
                      </option>
                       </c:forEach>
                 </select>
            </li>
              <li><input name="" type="submit" class="scbtn" value="查询" /></li>
```

```
            </ul>
        </form>
        <table class="tablelist" >
            <thead>
            <tr>
          <th style="width:100px;text-align:center;">新闻标题</th>
             <th style="width:100px;text-align:center;">新闻类别</th>
             <th style="width:100px;text-align:center;">发布/更新时间</th>
             <th style="width:100px;text-align:center;">发布状态</th>
             <th style="width:200px;text-align:center;">操作</th>
             </tr>
             </thead>
                <tbody>
                <c:if test="${!empty pb.list}">
                    <c:forEach items="${pb.list}" var="news">
                        <tr>
                            <td align="center">${news.title}</td>
                            <td align="center">${news.categoryName}</td>
                            <td align="center">
         <fmt:formatDate value="${news.publishTime}" pattern="yyyy-MM-dd HH:mm:ss"/>
                          </td>
                             <td align="center">
                               <c:if test="${news.publishStatus=='1'}" var="flag">
                                 已发布
                             </c:if>
                              <c:if test="${not flag}">
                                 已撤稿
                             </c:if>
                            </td>
                          <td align="center">
        <a      href="${pageContext.request.contextPath}/toEditNews.action?newsId=${news.newsId}">
        修改
       </a>
            <c:if test="${news.publishStatus=='2'}" var="status_flag">
                 | <a href='#' onclick="upNews(${news.newsId})">发布</a>
            </c:if>
             <c:if test="${not status_flag}">
                     | <a href='#' onclick="downNews(${news.newsId})">撤稿</a>
            </c:if>
             | <a href='#' onclick="delNews(${news.newsId})">删除</a>
                                 </td>
                        </tr>
                    </c:forEach>
                    <tr>
                    <td colspan="9" align="center">
```

```
                        <div class="pagination">
                        第${pb.currentPage}页
                                共${pb.totalPage}页
                                共${pb.count}条新闻
                         <div style="height:10px;"></div>
                         <c:if test="${pb.currentPage==1}"> 首页  上一页 </c:if>
                             <c:if test="${pb.currentPage>1}">
                                  <a href='#' onclick="fy(1)">首页</a>
                                  <a href='#' onclick="fy(${pb.currentPage-1})">上一页</a>
                             </c:if>
                             <c:if test="${pb.currentPage<pb.totalPage}">
                                  <a href='#'  onclick="fy(${pb.currentPage+1})">下一页</a>
                                       <a  href='#' onclick="fy(${pb.totalPage})">尾页</a>
                             </c:if>
                             <c:if  test="${pb.currentPage==pb.totalPage}">
                                 下一页  尾页</c:if>   
                             跳转至 <input type="text" style="height:22px;width:30px;"
                   value="${gotoPageNo}" name="gotoPageNo" id="gotoPageNo"> 页
                           <a href='#' onclick="validate()">跳转</a>
                   </div>
                        </td>
                        </tr>
                   </c:if>
              <c:if test="${empty pb.list}">
                   <div>
                        <tr>
                        <td colspan="5" align="center">
                        暂无新闻
                        </td>
                        </tr>
                   </div>
              </c:if>
           </tbody>
        </table>
      <script type="text/javascript">
        //翻页
         function fy(gotoPageNo)
           {
               var form=document.getElementById("newsListForm");
           form.action=
"${pageContext.request.contextPath}/findNewsByPage.action?currentPage=" + gotoPageNo ;
               form.submit();
           }
        //跳转验证
        function validate()
           {
```

```
            var totalPage=${pb.totalPage};
             var gotoPageNo = document.getElementById("gotoPageNo").value;
             if(gotoPageNo >totalPage || gotoPageNo <= 0 )
             {
                alert("你输入的页码有误！");
              }else{
                fy(gotoPageNo)
            }
          }
        //新闻上架（发布）
        function upNews(newsId){
            $.ajax({
                url:"${pageContext.request.contextPath }/setNewsPublishStatus.action",
                type:"post",
                //data 表示发送的数据
                data:JSON.stringify({newsId:newsId,publishStatus:1}),
                // 定义发送请求的数据格式为 JSON 字符串
                contentType:"application/json;charset=UTF-8",
                //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
                dataType:"json",
                //成功响应的结果
                success:function(data){
                    if(data!=null){
                        if(data.newsId>0){
                            alert("发布成功！");
                            window.location.reload();
                        }else{
                            alert("发布失败！");
                            window.location.reload();
                        }
                    }
                }
                });
        }
        //新闻下架（撤稿）
        function downNews(newsId){
            $.ajax({
                url:"${pageContext.request.contextPath }/setNewsPublishStatus.action",
                type:"post",
                //data 表示发送的数据
                data:JSON.stringify({newsId:newsId,publishStatus:2}),
                // 定义发送请求的数据格式为 JSON 字符串
                contentType:"application/json;charset=UTF-8",
                //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
                dataType:"json",
                //成功响应的结果
```

```
                success:function(data){
                    if(data!=null){
                        if(data.newsId>0){
                            alert("撤稿成功！");
                            window.location.reload();
                        }else{
                            alert("撤稿失败！");
                            window.location.reload();
                        }
                    }
                }
                });
        }
        //删除新闻
        function delNews(newsId){
            if(window.confirm("您确定要删除吗？"))
            {
                $.ajax({
                    url:"${pageContext.request.contextPath }/delNews.action",
                    type:"post",
                    //data 表示发送的数据
                    data:JSON.stringify({newsId:newsId}),
                    // 定义发送请求的数据格式为 JSON 字符串
                    contentType:"application/json;charset=UTF-8",
                    //定义回调响应的数据格式为 JSON 字符串，该属性可以省略
                    dataType:"json",
                    //成功响应的结果
                    success:function(data){
                        if(data!=null){
                            if(data.newsId>0){
                                alert("删除成功！");
                                window.location.reload();
                            }else{
                                alert("删除失败！");
                                window.location.reload();
                            }
                        }
                    }
                });
            }
        }
       </script>
      </div>
      <script type="text/javascript">
          $('.tablelist tbody tr:odd').addClass('odd');
        </script>
```

```
</body>
</html>
```

新闻列表页面 news_list.jsp 页面文件主要用来展示新闻列表，通过访问 NewsController 类的 findNewsByPage()可以跳转到 news_list.jsp 页面并在页面中迭代显示新闻列表信息，并在此基础上可以通过链接对用户进行修改、删除、发布/撤稿等相关操作。

文件 16.34　add_news.jsp

```
<!-- 此处省略部分代码 -->
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!-- 此处省略部分代码 -->
      <form action="${pageContext.request.contextPath}/addUser.action" name="ff"
        method="post" onsubmit="return checkValue()">
<!-- 此处省略部分代码 -->
    新闻标题：
     <input id="title" name="title" value="${news.title}" type="text" />
    新闻类别：
       <select name="categoryId" id="categoryId" >
         <option value="">--请选择--</option>
         <c:forEach items="${categoryList}" var="c">
           <option value="${c.categoryId}"
             <c:if test="${c.categoryId eq news.categoryId}">
                selected="selected"
             </c:if>>
             ${c.categoryName}
           </option>
         </c:forEach>
     </select>
     新闻内容页标题：
  <input id="contentTitle" name="contentTitle" value="${news.contentTitle}" type="text"/>
       内容摘要：
      <textarea id="contentAbstract" name="contentAbstract" cols="100" rows="2"
           style="width:800px;height:50px;" >${news.contentAbstract}</textarea>
     内容：
       <textarea id="content" name="content" cols="100" rows="4"
          style="width:800px;height:100px;" >${news.content}</textarea>
     关键词：
    <input id="keywords" name="keywords"  value="${news.keywords}" type="text"/>
     作者/来源：
   <input id="author" name="author"  value="${news.author}" type="text" />
   <input name="" type="submit" value="确认发布"/>
      <input name="" type="button" onclick="goback();" value="返回列表"/>
    </form>
 <script>
    function goback(){
     window.location.href="${pageContext.request.contextPath}/findNewsByPage.action";
```

```
    }
    function checkValue() {
     var str = document.getElementById("title").value;
     if (str.length <1) {
         alert("新闻标题");
         document.getElementById("title").focus();
         return false;
     }
     str = document.getElementById("categoryId").value;
     if (str == '') {
         alert("请选择新闻类别");
         return false;
     }
     str = document.getElementById("contentTitle").value;
     if (str.length <1) {
         alert("新闻内容页标题");
         document.getElementById("contentTitle").focus();
         return false;
     }
     str = document.getElementById("content").value;
     if (str.length <1) {
         alert("请输入新闻内容");
         document.getElementById("content").focus();
         return false;
     }
     return true;
    }
</script>
<!-- 此处省略部分代码 -->
```

文件 16.35 edit_news.jsp

```
<!-- 此处省略部分代码 -->
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!-- 此处省略部分代码 -->
      <form action="${pageContext.request.contextPath}/addUser.action" name="ff"
        method="post" onsubmit="return checkValue()">
     <input id="newsId" name="newsId" value="${news.newsId}" type="hidden" />
<!-- 此处省略部分代码 -->
    新闻标题:
     <input id="title" name="title" value="${news.title}" type="text" />
    新闻类别:
      <select name="categoryId" id="categoryId" >
        <option value="">--请选择--</option>
        <c:forEach items="${categoryList}" var="c">
          <option value="${c.categoryId}"
            <c:if test="${c.categoryId eq news.categoryId}">
```

```
                selected="selected"
              </c:if>>
              ${c.categoryName}
            </option>
          </c:forEach>
      </select>
      新闻内容页标题：
    <input id="contentTitle" name="contentTitle" value="${news.contentTitle}" type="text"/>
        内容摘要：
        <textarea id="contentAbstract" name="contentAbstract" cols="100" rows="2"
              style="width:800px;height:50px;" >${news.contentAbstract}</textarea>
      内容：
        <textarea id="content" name="content" cols="100" rows="4"
            style="width:800px;height:100px;" >${news.content}</textarea>
      关键词：
      <input id="keywords" name="keywords"  value="${news.keywords}" type="text"/>
      作者/来源：
     <input id="author" name="author"  value="${news.author}" type="text" />
     <input name="" type="submit" value="确认修改"/>
        <input name="" type="button" onclick="goback();" value="返回列表"/>
      </form>
   <script>
      function goback(){
      window.location.href="${pageContext.request.contextPath}/findNewsByPage.action";
      }
      function checkValue() {
       var str = document.getElementById("title").value;
       if (str.length <1) {
            alert("新闻标题");
            document.getElementById("title").focus();
            return false;
       }
       str = document.getElementById("categoryId").value;
       if (str == '') {
            alert("请选择新闻类别");
            return false;
       }
       str = document.getElementById("contentTitle").value;
       if (str.length <1) {
            alert("新闻内容页标题");
            document.getElementById("contentTitle").focus();
            return false;
       }
       str = document.getElementById("content").value;
       if (str.length <1) {
            alert("请输入新闻内容");
```

```
            document.getElementById("content").focus();
            return false;
        }
        return true;
      }
  </script>
  <!-- 此处省略部分代码 -->
```

步骤02 启动项目，测试新闻管理。将项目发布到 Tomcat 服务器并启动，访问地址：http://localhost:8080/news_publish/login.jsp，进入登录页面，以信息员身份登录后进入主界面，如图 16.11 所示。



图 16.11 信息员系统后台界面

信息员登录系统后台后，可以对新闻进行相关操作。如图 16.12、图 16.13、图 16.14 所示的界面中，信息员可以进行查询新闻、删除新闻、发布/撤稿新闻、修改新闻等操作。其中，查询新闻实现了分页效果。另外，发布新闻和修改新闻中，新闻内容输入用的是多行文本框，在实际项目开发中可以用“在线编辑器”代替，以便以富媒体形式编辑和发布新闻内容。



图 16.12 查询新闻界面



图 16.13　发布新闻界面



图 16.14　修改新闻界面

从以上测试效果可以看出，已实现了新闻管理模块的相关功能。

16.7　登录验证

虽然在 16.5 节中已经实现了用户登录功能，但是此功能并不完善。假设在其他控制器类中也包含一个访问用户管理或新闻管理页面的方法，那么用户完全可以绕过登录步骤，而直接通过访问该

方法的方式进入用户管理或新闻管理页面。让未登录的用户直接访问后台管理页面是十分不安全的。为了避免这种情况的发生，并提升系统的安全性，可以创建一个登录拦截器来拦截所有请求。只有已登录用户的请求才能够通过，而对于未登录用户的请求，系统会将请求转发到登录页面，并提示用户登录。

16.7.1 创建登录拦截器类

在 src 目录下创建一个 com.ssm.interceptor 包，并在包中创建登录拦截器类 LoginInterceptor 来实现用户登录的拦截功能，代码如文件 18.36 所示。

文件 18.36 LoginInterceptor.java

```
package com.ssm.interceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
import com.ssm.po.User;
public class LoginInterceptor implements HandlerInterceptor{
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
                         Object handler) throws Exception {
     //获取请求的 URL
     String url=request.getRequestURI();
     //除了登录请求外，其他的 URL 都进行拦截控制
     if(url.indexOf("/login.action")>=0){
         return true;
     }
     //获取 Session
     HttpSession session=request.getSession();
     User user=(User)session.getAttribute("login_user");
     //判断 Session 中是否有用户数据，
         //如果有，就返回 true，表示用户已登录，继续向下执行
     if(user!=null){
         return true;
     }
     //不符合条件，给出提示信息，并转发到登录页面
     request.setAttribute("msg", "您还没有登录，请先登录！");
    request.getRequestDispatcher("/WEB-INF/jsp/skip.jsp").forward(request, response);
     return false;
    }
    @Override
    public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1,
                          Object arg2, Exception arg3)   throws Exception { }
    @Override
```

```
    public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1,
                    Object arg2, ModelAndView arg3) throws Exception { }
}
```

在文件 18.36 的 preHandle()方法中，首先获取了用户 URL 请求，然后通过请求来判断是否为用户登录操作，只有对用户登录的请求才不进行拦截。接下来获取了 Session 对象，并获取 Session 中的用户信息。如果 Session 中的用户信息不为空，就表示用户已经登录，拦截器将放行；如果 Session 中的用户信息为空，就表示用户未登录，系统会转发到登录页面，并提示用户登录。

16.7.2　配置拦截器

在 springmvc-config.xml 文件中配置登录拦截器信息，其配置代码如下。

```
<!-- 配置拦截器 -->
    <mvc:interceptors>
    <!-- 使用 bean 直接定义在<mvc:interceptors>下面的 Interceptor 将拦截所有请求 -->
        <!-- <bean class="com.ssm.interceptor.LoginInterceptor" /> -->
        <mvc:interceptor>
            <!-- 配置拦截器作用的路径 -->
            <mvc:mapping path="/**" />
            <!-- 配置不需要拦截器作用的路径（前台页面需要访问控制器的路径） -->
            <mvc:exclude-mapping path="/index.action"/>
            <mvc:exclude-mapping path="/findNewsByCategoryIdPage.action"/>
            <mvc:exclude-mapping path="/findFrontNewsByNewsId.action"/>
            <bean class="com.ssm.interceptor.LoginInterceptor" />
        </mvc:interceptor>
    </mvc:interceptors>
```

上述配置代码会将所有的用户请求都交由登录拦截器来处理。当然，也可以通过<mvc:exclude-mapping path="">配置不需要拦截器作用的路径，例如<mvc:exclude-mapping path="/index.action"/>是前台首页需要访问控制器 NewsController 的路径。至此，登录拦截器的实现工作就已经完成。未登录的用户执行访问后台的管理页面方法会由拦截器转发到系统登录页面，并登录窗口中给出提示信息，如图 16.15 所示。



图 16.15　转发到登录页面

注　意

作为一个新闻管理系统，除了后台以外，还要实现前台新闻的展示（新闻首页、新闻列表页和新闻详细内容显示页等），可以使用 SSM 框架知识和 Bootstrap、jQuery 框架相关知识进行实现，因前端知识不是本书的重点内容，本章不再做详细介绍，读者可以参考本章案例代码。

16.8 项目小结

本章主要通过一个新闻管理系统讲解 SSM 框架的整合使用。首先对系统的功能、结构等进行简单的介绍；然后对系统数据库表进行分析和设计；接着详细地讲解了系统的环境搭建工作；最后详细地讲解了系统用户管理模块、新闻管理模块和登录验证的实现。通过本章的学习，读者可以熟练地掌握 SSM 框架的整合使用，并能熟练地使用 SSM 框架实现系统功能模块的开发工作。本系统是 SSM 框架综合使用的案例，读者要多加练习，并熟练编写各个功能模块的实现代码，这样才能将前面所学的知识融会贯通。